Doctoral Dissertation

Improving the Performance and Energy Efficiency of GPGPU Computing through Adaptive Cache and Memory Management Techniques

Kyu Yeun Kim

Department of Computer Science and Engineering

Graduate School of UNIST

2020

Improving the Performance and Energy Efficiency of GPGPU Computing through Adaptive Cache and Memory Management Techniques

Kyu Yeun Kim

Department of Computer Science and Engineering

Graduate School of UNIST

Improving the Performance and Energy Efficiency of GPGPU Computing through Adaptive Cache and Memory Management Techniques

A dissertation
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Kyu Yeun Kim

1. 3. 2020
Approved by

Advisor
Woongki Baek

Improving the Performance and Energy Efficiency of GPGPU Computing through Adaptive Cache and Memory Management Techniques

Kyu Yeun Kim

This certifies that the dissertation of Kyu Yeun Kim is approved.

1. 3. 2020

signature

Advisor : Woongki Baek

signature

Young-ri Choi : Committee Member #1

signature

Jongeun Lee : Committee Member #2

signature

Sam H. Noh : Committee Member #3

signature

Kyuho Lee : Committee Member #4

Abstract

As the performance and energy efficiency requirement of GPGPUs have risen, memory management techniques of GPGPUs have improved to meet the requirements by employing hardware caches and utilizing heterogeneous memory. These techniques can improve GPGPUs by providing lower latency and higher bandwidth of the memory. However, these methods do not always guarantee improved performance and energy efficiency due to the small cache size and heterogeneity of the memory nodes. While prior works have proposed various techniques to address this issue, relatively little work has been done to investigate holistic support for memory management techniques.

In this dissertation, we analyze performance pathologies and propose various techniques to improve memory management techniques. First, we investigate the effectiveness of advanced cache indexing (ACI) for high-performance and energy-efficient GPGPU computing. Specifically, we discuss the designs of various static and adaptive cache indexing schemes and present implementation for GPGPUs. We then quantify and analyze the effectiveness of the ACI schemes based on a cycle-accurate GPGPU simulator. Our quantitative evaluation shows that ACI schemes achieve significant performance and energy-efficiency gains over baseline conventional indexing scheme. We also analyze the performance sensitivity of ACI to key architectural parameters (i.e., capacity, associativity, and ICN bandwidth) and the cache indexing latency. we also demonstrate that ACI continues to achieve high performance in various settings.

Second, we propose IACM, integrated adaptive cache management for high-performance and energy-efficient GPGPU computing. Based on the performance pathology analysis of GPGPUs, we integrate state-of-the-art adaptive cache management techniques (i.e., cache indexing, bypassing, and warp limiting) in a unified architectural framework to eliminate performance pathologies. Our quantitative evaluation demonstrates that IACM significantly improves the performance and energy efficiency of various GPGPU workloads over the baseline architecture (i.e., 98.1% and 61.9% on average, respectively) and achieves considerably higher performance than the state-of-the-art technique (i.e., 361.4% at maximum and 7.7% on average). Furthermore, IACM delivers significant performance and energy efficiency gains over the baseline GPGPU architecture even when enhanced with advanced architectural technologies (e.g., higher capacity, associativity).

Third, we propose bandwidth- and latency-aware page placement (BLPP) for GPGPUs with heterogeneous memory. BLPP analyzes the characteristics of a application and determines the optimal page allocation ratio between the GPU and CPU memory. Based on the optimal page allocation ratio, BLPP dynamically allocate pages across the heterogeneous memory nodes. Our

experimental results show that BLPP considerably outperforms the baseline and state-of-the-art technique (i.e., 13.4% and 16.7%) and performs similar to the static-best version (i.e., 1.2% difference), which requires extensive offline profiling.

Contents

List of Figures

List of Tables

I Introduction

From big data computing to machine learning, GPGPUs are being used in various computing domains as it utilizes thousands of cores for high performance and energy-efficient computing. To further improve the performance and energy efficiency, modern GPGPUs have begun to improve memory management techniques in terms of both architecture and system software. For the GPGPU architecture, hardware caches have been widely adopted in the memory hierarchy [1, 2] for faster memory access. For example, the NVIDIA Kepler GK110 architecture includes 16KB (or 48KB) L1 cache per core and total capacity of 1,536KB L2 caches [1]. The rationale behind this design decision is that small yet fast GPGPU caches would effectively capture the locality in the memory accesses of GPGPU workloads and reduce memory access overhead, similarly to CPU caches.

However, incorporating hardware caches in the GPGPU memory hierarchy does not always guarantee enhanced performance and energy efficiency in GPGPU computing. The fundamental limitation is that in GPGPU, thousands of GPGPU threads share significantly small capacity of GPGPU caches, which can cause various performance pathologies. Without effective cache management, GPGPU architectures may fail to achieve the best possible performance and energy efficiency when using GPGPU caches.

In order to improve effectiveness of hardware caches, researchers have proposed architectural techniques based on adaptive warp scheduling and limiting [3–8], and cache bypassing [4, 5]. Relatively little work, however, has been done in the context of advanced cache indexing (ACI) for GPGPUs, which has been shown to be one of the most effective techniques to improve performance in CPU hardware caches [9–11].

To bridge this gap, we investigate the effectiveness of advanced cache indexing for high performance and energy efficient GPGPU computing [12, 13]. We discuss the design of various static and adaptive cache indexing schemes and present implementation for GPGPU architectures. We then quantitatively evaluate the static and adaptive cache indexing schemes compared to the baseline conventional indexing scheme in terms of performance and energy efficiency using GPGPU workloads. We also investigate the performance sensitivity of the advanced cache indexing schemes to key architectural parameters such as cache capacity and indexing latency.

Furthermore, little work has been done to create a unified architecture framework that tightly integrates the state-of-the-art adaptive cache management techniques and investigate their effectiveness when they are tightly integrated. Based on the analysis of GPGPU performance pathologies, we conclude that multiple cache management schemes needs to be integrated to achieve the best possible performance. Therefore, we propose *IACM* [14, 15], integrated adaptive cache management for high-performance and energy-efficient GPGPU computing. IACM incorporates the state-of-the-art adaptive cache management techniques (i.e., adaptive cache indexing, adaptive warp limiting, and cache bypassing) in a unified architectural framework. Based on a cycle-accurate GPGPU simulator [16] and various GPGPU workloads, we quantify

the performance and energy efficiency of IACM.

In terms of system software, GPGPUs have begun to utilize heterogeneous memory for improved performance. Researchers [17–23] have presented the design and implementation of virtual memory for GPGPUs with heterogeneous memory that comprises the GPU (e.g., GDDR5) and CPU (e.g., DDR4) memory nodes. By using both GPU and CPU memory nodes, GPGPUs are able to increase effective bandwidth and capacity of the memory. For better heterogeneous memory system, system must provide the transparent support for virtual memory so that programmers can fully utilize all the available heterogeneous memory nodes without the need for manually managing the data transfers.

In order to achieve the best possible performance on GPGPUs with heterogeneous memory, system should first dynamically categorize application characteristics. Then, it should allocate pages judiciously across the heterogeneous memory nodes by considering the application characteristics and the differences between GPU and CPU memory in terms of both bandwidth and latency. Most of the aforementioned prior works have only investigated the efficient design and implementation of the address translation [20,21], cache hierarchy [19], and warp scheduling [18], lacking the heterogeneity-aware memory management.

The prior work in GPGPUs with heterogeneous memory [17] has proposed a memory management technique that places memory pages by considering the memory bandwidth difference of the GPU and CPU memory. However, it lacks the consideration of the GPGPU caches, which significantly affect the memory performance in terms of bandwidth and latency. This leads to allocating pages across the heterogeneous memory nodes in a latency-oblivious manner, achieving suboptimal performance.

To bridge this gap, we propose bandwidth- and latency-aware page placement (BLPP) for GPGPUs with heterogeneous memory which consists of three phases [24]. First, BLPP collects the performance counter data of the target application using offline profile data or using runtime information. Second, BLPP determines the optimal memory allocation ratio based on the application characteristics and the performance differences of the heterogeneous memory nodes. Finally, it dynamically allocates memory pages based on the optimal allocation ratio. We demonstrate the effectiveness of BLPP through quantitative evaluation with various GPGPU workloads.

Specifically, this dissertation makes the following contributions:

- Analysis on Advanced Cache Indexing Schemes
  - We explain performance pathologies of GPGPU hardware cache by analyzing memory access pattern of GPGPU application and discuss various static and adaptive cache indexing schemes that can mitigate the issues. We discuss design and implementations of advanced cache indexing schemes for high performance and energy-efficient GPGPU computing.

  - With a cycle-accurate GPGPU simulator [16], we provide a quantitative evaluation of the advanced cache indexing schemes. We use benchmarks from various benchmark suites with a wide range of memory access characteristics for in-depth analysis. Specifically, we quantify the performance and energy efficiency of the advanced cache indexing schemes for the L1 data and L2 caches of the GPGPU architecture. We provide various data such as cycle breakdown, reservation fails, and miss rate to gain a deeper insight.
  - We investigate the performance sensitivity of the advanced cache indexing schemes to key architectural parameters: indexing latency, number of sets, associativity, and capacity of L1 data cache. Our sensitivity study demonstrates that the advanced cache indexing schemes continues to provide significant performance gains even when the additional indexing latency occurs due to the hardware complexity. And even when the advanced cache indexing schemes are used in a hardware cache that is enhanced with larger capacity or high associativity, it will continue to provide significant performance gains.

- Propose and Evaluate Integrated Adaptive Cache Management

  - We propose IACM, integrated adaptive cache management for high performance and energy-efficient GPGPU computing. IACM is an integrated GPGPU architecture that incorporates the state-of-the-art adaptive cache management techniques (i.e., adaptive cache indexing, adaptive warp limiting, and cache bypassing) in an unified manner.
  - We present three IACM designs, each with a different methods of unifying adaptive cache management techniques. We perform extensive design parameter sweeps to find parameters with the best performance for each of three IACM designs based on a cycle-accurate GPGPU simulator. We then compare performance of three IACM designs and determine the design that provides the highest performance among the three.
  - We quantify the performance and energy efficiency of IACM using the IACM design with the best performance. Our quantitative evaluation demonstrates that IACM significantly outperforms the baseline GPGPU architecture in terms of performance and energy efficiency (i.e., 98.1% and 61.9% on average) by effectively unifying the adaptive cache management techniques in an integrated manner and eliminating the performance pathologies.
  - We quantitatively compare IACM with the state-of-the-art technique [6] which also integrates various cache management techniques. Our experimental results show that IACM outperforms the state-of-the-art technique with a majority of the evaluated benchmarks. This is because IACM employs the adaptive cache indexing technique

unlike state-of-the-art which employs the static cache indexing technique, and applies the adaptive techniques in a more coordinated manner. Overall, IACM achieves considerably higher performance (i.e., 361.4% at maximum and 7.7% on average) than the state-of-the-art technique across all the 20 evaluated benchmarks, which clearly demonstrate the effectiveness of IACM.

- Propose and Evaluate Bandwidth- and Latency-aware Page Placement
  - We identify and quantify that the state-of-the-art page placement technique [17] achieves suboptimal performance on the practical GPGPU architecture with heterogeneous memory. We analyze that this is mainly because the prior technique lacks the consideration of the performance effects of the GPGPU caches in terms of bandwidth and latency.
  - We propose BLPP, bandwidth- and latency-aware page placement for GPGPUs with heterogeneous memory. BLPP is comprised of three phases. First, BLPP collects the performance characteristics (e.g., cache miss rates and memory traffic) of the target application using offline profile data or runtime information. Second, it determines the optimal allocation ratio across the heterogeneous memory nodes based on the application characteristics. Finally, it dynamically places memory pages by using the optimal allocation ratio determined from previous phase.
  - We propose and evaluate two versions of BLPP. The static version of BLPP (S-BLPP) employs the offline profile data of the target application. In contrast, the dynamic version of BLPP (D-BLPP) dynamically classifies the target application characteristics and determines the optimal allocation ratio without requiring offline profiling. Using quantitative evaluation, we show that D-BLPP achieves similar performance to S-BLPP.
  - We quantify the effectiveness of BLPP based on a cycle-level GPGPU simulator and a variety of GPGPU workloads. Our experimental results show that BLPP considerably outperforms (e.g., 16.7% higher performance) the state-of-the-art technique [17] and achieves the performance similar (e.g., 1.2% lower performance) to that of the static best version, which requires extensive offline profiling for every application, dataset, and memory allocation ratio. We also provide execution cycle breakdown and memory allocation ratio data of each techniques for an in-depth analysis.

The rest of this dissertation is organized as follows. Section II provides background information on the baseline GPGPUs and its performance pathologies. Section III provides performance and energy efficiency analysis on GPGPUs when using advanced cache indexing schemes. Section IV presents the IACM architecture and compares its performance and energy efficiency against the baseline and state-of-the-art architecture. Section V presents BLPP and compares

its performance to the baseline and state-of-the-art technique. Section VI summarizes related works and Section VII concludes the dissertation.

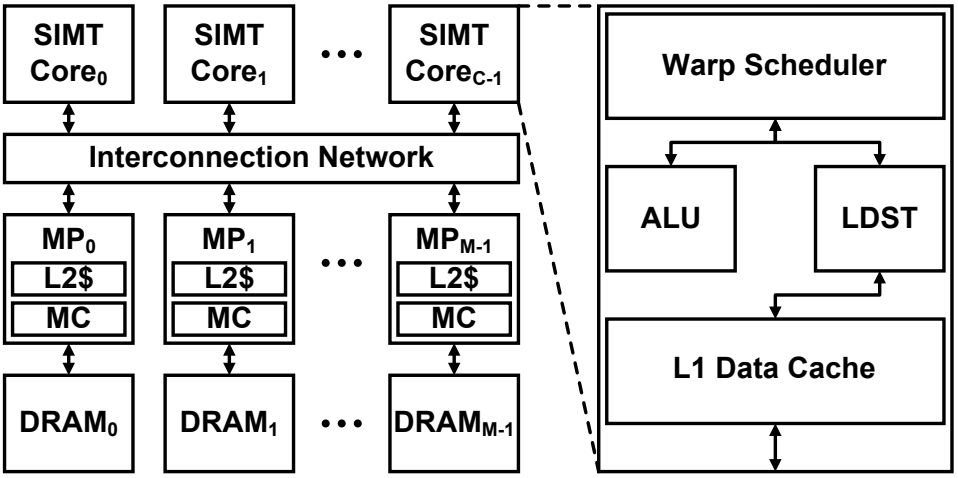


Figure 1: The baseline GPGPU architecture

II Background

2.1 Baseline GPGPU Architecture

Figure 1 shows the baseline GPGPU architecture that consists of multiple single-instruction multiple-thread (SIMT) cores[1]. Each SIMT core includes L1 memory components such as a private L1 data cache, which serves memory requests from LDST unit. When a cache miss occurs from L1 data cache, memory request is sent to the corresponding memory partition through the interconnection network. Each memory partition consists of hardware components such as an L2 cache bank, and a memory controller and is connected to an off-chip DRAM module. To support outstanding memory requests, L1 and L2 caches are augmented with the miss status holding registers (MSHRs), which dynamically track the status of pending misses.

A GPGPU program consists of kernels that are offloaded to GPGPUs. These kernels can be divided into concurrent thread blocks (CTA), warps, and threads. Threads belonging to the same CTA can be synchronized using synchronization primitives such as barriers and share data through a shared memory. With CTA being consisted of multiple warps, group of threads form a single warp. SIMT core executes threads in a warp granularity, and every thread within a scheduled warp executes in a lockstep manner. Execution order of warps can affect overall performance of GPGPU. Therefore, prior works have extensively investigated warp scheduling techniques such as Greedy-Then-Oldest (GTO) [3] and two-level [8] scheduling techniques.

2.2 Performance Pathologies of GPGPU Caches

Similar to CPU caches, GPGPU caches may suffer from well-known performance pathologies such as thrashing and contention. In GPGPU computing, cache thrashing occurs when the working-set size of a kernel exceeds the capacity of the cache. One of the key parameters that

[1]MP and MC in Figure 1 denote memory partition and memory controller, respectively.

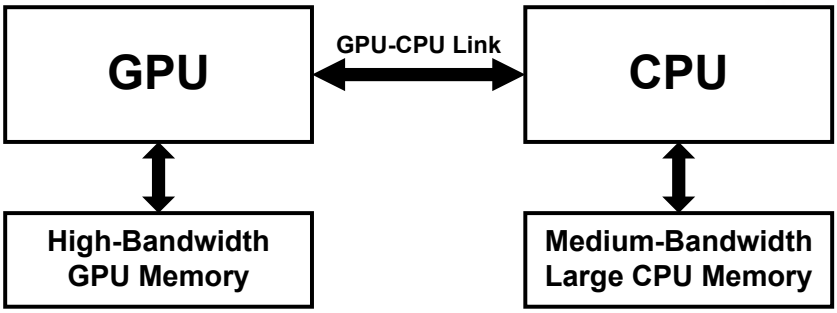


Figure 2: GPGPU with heterogeneous memory

determines the effective working-set size of a kernel is number of active warps, which is controlled by the warp scheduler. GPGPUs are significantly more vulnerable to cache thrashing than CPUs due to thousands of threads sharing the cache.

Cache contention occurs when concurrent threads compete for hardware resources required for a cache operation. In GPGPU computing, the hardware resources that often trigger drastic performance degradation from contention are cache lines, MSHR entries, and miss queue entries. A *reservation fail* occurs when a memory operation cannot proceed because of a failure to reserve any of the aforementioned hardware resources.

Specifically, a cache-line reservation fail for a memory operation occurs when all the cache lines in the set associated with the memory operation are currently reserved for other pending memory operations. With ineffective cache indexing, cache-line reservation fails can frequently occur even when there are plenty of available cache lines. Reservation fails for MSHR and miss queue entries typically occur with frequent cache misses.

Threads that cause cache contention or cache thrashing can belong to a same warp or different warps. Following the terminologies used in [3], we refer to the former as the intra-warp interference and the latter as the inter-warp interference. Reason we are distinguishing intra-warp and inter-warp interferences is because different cache management technique should be used to mitigate performance pathologies depending on the interference, which will be discussed in the later sections.

2.3 Heterogeneous Memory Systems

To date, high-performance GPGPUs have been mostly managed as separate computing devices from CPUs, requiring GPGPU programmers to manually manage memory transfer between the GPU and CPU memory. Therefore, programmers had to decide which and when memories should be transferred between the GPU and CPU memory for optimal performance. To enhance the programmability and support a wide range of applications, researchers have extensively investigated automatic memory management techniques to provide the unified address space for GPGPUs with heterogeneous memory that comprises the GPU and CPU memory nodes [17, 19, 20, 22, 23, 25].

Figure 2 shows the GPGPU architecture with heterogeneous memory, which enables the GPGPU to seamlessly access the data in the GPU and CPU memory nodes without manual memory management by a programmer. In line with the state-of-the-art designs, we assume that the CPU and GPGPU are connected via a dedicated high-performance GPU-CPU link such as NVIDIA's NVLINK [26] and AMD's HyperTransport [27].

Prior works have presented the design and implementation of the OS-controlled unified memory for GPGPUs with heterogeneous memory [19, 20, 25]. In this work, we follow the work [19] and assume that the underlying GPGPU architecture with heterogeneous memory implements the *selective caching* protocol. Reason we are implementing selective caching is mainly because the selective caching protocol achieves high performance with low hardware complexity. In contrast, the other proposals require intrusive modifications in the GPU and/or CPU memory hierarchy [20, 25], significantly increasing the hardware complexity.

The selective caching protocol disallows the GPGPU caching of any data that is (1) mapped in the CPU physical memory or (2) actively used by the CPU on-chip caches. By enforcing these properties, the selective caching protocol eliminates the need for supporting cache coherence in the GPGPU caches and requires no or little modifications in the GPU and CPU memory hierarchy [19].

Listing 1: The code snippet of the ATAX benchmark

```
void atax_kernel1(float *A, float *x, float *tmp)
{
 // blockDim.x = 256
 int i = blockId.x*blockDim.x + threadId.x;
 if (i < NX) {
  int j;
  // NY = 4096
  for(j=0; j < NY; j++)
   tmp[i] += A[i * NY + j] * x[j];
 }
}
```

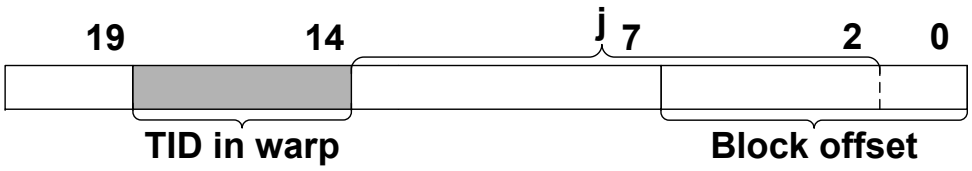


Figure 3: An example of cache conflicts when running ATAX

III Quantifying the Performance and Energy Efficiency of Advanced Cache Indexing for GPGPU Computing

3.1 Motivation

The baseline GPGPU architecture is potentially vulnerable to cache thrashing and contention given that thousands of threads share small hardware caches. To gain a deeper understanding of the cache performance pathology of GPGPUs, we present a case study with the ATAX benchmark in the PolyBench benchmark suite [28], which contains a pathological memory access pattern.

Listing 1 shows the code snippet of ATAX. Because NY is 4096, the memory addresses for concurrent accesses to array A performed by all 32 threads within a warp have unique bit values from B_{14} to B_{18}, which essentially encode the thread ID in a warp (see Figure 3). Thus, the bits from B_{14} to B_{18} are highly effective for avoiding reservation fails and conflict misses among the threads within a warp.[2] In contrast, because every thread in a warp executes in a lock-step manner, the bits from B_7 to B_{13}, determined by the variable j, have identical values across all threads within a warp. Therefore, the bits from B_7 to B_{13} are ineffective if used to mitigate the cache thrashing among the threads.

[2]Note that a reservation fail occurs when there is no available cache line or hardware resource (MSHR or miss queues) for a memory request because all of them are currently in use to serve other pending memory requests.

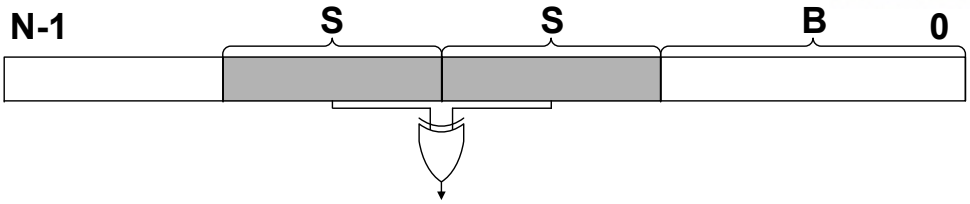


Figure 4: Bitwise XOR indexing

With a hardware cache that consists of 32 sets (i.e., indexing bits consist of five bits), the conventional cache indexing scheme employs the bits from B_7 to B_{11}, causing drastic cache contention among the threads in the same warp. If the cache employs an advanced cache indexing scheme that can avoid this cache contention, the overall performance can be significantly improved. For instance, adaptive cache indexing can indicate that the bits from B_{14} to B_{18} are highly effective for avoiding cache contention based on runtime information, therefore using employ them as the indexing bits.

The cache thrashing and contention of GPGPU workloads can be triggered by *intra-warp* interference and *inter-warp* interference [3]. While inter-warp interference can be mitigated using other GPGPU cache management techniques, such as warp scheduling [3], they are generally ineffective when used to address intra-warp interference. Because advanced cache indexing is expected to efficiently address both types of interference, it is crucial to thoroughly investigate the effectiveness of various advanced cache indexing schemes for high performance and energy efficient GPGPU computing.

3.2 Static Cache Indexing

This section discusses advanced static cache indexing schemes. In Sections 3.2 and 3.3, without a loss of generality, we assume a GPGPU architecture with an N-bit address space. We also assume a hardware cache whose block size, associativity, and number of sets are correspondingly 2^B bytes, W, and 2^S. An N-bit memory address A is expressed as $A[N-1:0]$ and its cache-block address A_B is expressed as $A[N-1:B]$. In addition, cache index I for a memory address is expressed as $I[S-1:0]$.

3.2.1 Bitwise XOR Indexing

Bitwise XOR indexing (BXI) computes the set index mapped to a cache-block address by performing the bitwise XOR operation for the first and second lowest S bits of the cache-block address [10]. Lowest $2S$ bits are used to construct a set index because the lower bits change more frequently than the higher bits in a typical sequence of memory accesses, resulting in a more uniform distribution of the memory addresses during a time interval. Figure 4 shows how BXI constructs the indexing bits for an N bit memory address. Owing to its simple and rapid hardware implementation, BXI is applicable to L1 caches. In addition, the area overhead of BXI

$$I_5 = A_{30} \oplus A_{29} \oplus A_{28} \oplus A_{27} \oplus A_{24} \oplus A_{22} \oplus A_{18} \oplus A_{17} \oplus A_{12}$$
$$I_4 = A_{31} \oplus A_{29} \oplus A_{28} \oplus A_{27} \oplus A_{26} \oplus A_{23} \oplus A_{21} \oplus A_{17} \oplus A_{16} \oplus A_{11}$$
$$I_3 = A_{30} \oplus A_{28} \oplus A_{27} \oplus A_{26} \oplus A_{25} \oplus A_{22} \oplus A_{20} \oplus A_{16} \oplus A_{15} \oplus A_{10}$$
$$I_2 = A_{29} \oplus A_{27} \oplus A_{26} \oplus A_{25} \oplus A_{24} \oplus A_{21} \oplus A_{19} \oplus A_{15} \oplus A_{14} \oplus A_9$$
$$I_1 = A_{28} \oplus A_{26} \oplus A_{25} \oplus A_{24} \oplus A_{23} \oplus A_{20} \oplus A_{18} \oplus A_{14} \oplus A_{13} \oplus A_8$$
$$I_0 = A_{31} \oplus A_{30} \oplus A_{29} \oplus A_{28} \oplus A_{25} \oplus A_{23} \oplus A_{19} \oplus A_{18} \oplus A_{13} \oplus A_7$$

Figure 5: Polynomial modulus indexing

is expected to be low because it requires 5 XOR gates in the case of the 4-way 16KB L1 data cache with a block size of 128 bytes.

3.2.2 Reverse-Engineered Bitwise XOR Indexing

Recent work has reverse-engineered the NVIDIA GTX470 architecture and concluded that a variant of the bitwise XOR indexing scheme appears to be employed in its L1 data cache [29]. Specifically, as for the 4-way 16KB L1 data cache with a block size of 128 bytes, the reverse-engineered bitwise XOR indexing (RXI) scheme computes the indexing bits as follows – $I_4 = A_{19} \oplus A_{11}$, $I_3 = A_{17} \oplus A_{10}$, $I_2 = A_{15} \oplus A_9$, $I_1 = A_{14} \oplus A_8$, and $I_0 = A_{13} \oplus A_7$ [29].

The area overhead of RXI is expected to be low because RXI requires the same number of the XOR gates to compute the indexing bits for the same cache configuration as BXI does. The main difference between BXI and RXI is that RXI employs higher bits to construct some of the indexing bits (e.g., A_{19} for I_4). We conjecture that the design decision of RXI has been made mainly based on the observation that some of the higher bits can effectively reduce the cache contention for some GPGPU workloads. Section 3.4 quantitatively compares the performance and energy efficiency outcomes of BXI and RXI.

3.2.3 Polynomial Modulus Indexing

Polynomial modulus indexing (PLI) represents an N-bit memory address as a polynomial $A(x)$ in the Galois field of 2 (GF(2)) [6,7,9,10]. For instance, $A(x)$ for memory address 57 is expressed as $x^5 + x^4 + x^3 + 1$. Suppose that $P(x)$ is a polynomial whose order is S. $A(x)$ can be expressed as $A(x) = P(x) \cdot Q(x) + R(x)$ where $Q(x)$ and $R(x)$ are polynomials in GF(2) and the order of $R(x)$ is less than S. In such a case, $R(x)$ can be considered as the polynomial representation of the cache index for the memory address [6]. In other words, cache index $R(x)$ is computed as $R(x) = A(x) \bmod P(x)$.

Prior work holds that PLI can achieve the best permutation when $P(x)$ is an irreducible polynomial [9]. For example, we assume a cache with 64 sets (i.e., $S = 6$) and 128-byte blocks

(i.e., $B = 7$). We also assume $P(x) = x^6 + x + 1$, which is one of the irreducible polynomials whose order is 6. Figure 5 shows the indexing bits constructed by PLI for the aforementioned cache, whose configuration is identical to that of the L2 cache evaluated in Section 3.4.

PLI near-randomly interleaves consecutive memory addresses with various strides, even when the stride is a multiple of the number of cache sets (i.e., 2^S) [6,9]. However, prior work states that PLI is not resistant to all possible memory strides, potentially leading to suboptimal performance in pathological cases (e.g., when the stride is $2^S - 1$) [11]. Furthermore, because the hardware logic required to compute the index is rather complicated (e.g., trees of XOR gates) and is in the critical path, PLI may increase the hit time of the cache [30]. We investigate the performance impact of the additional indexing latency in Section 3.4. The area overhead of PLI is expected to be rather low (but slightly higher than BXI and RXI) because it requires tens of XOR gates to compute the indexing bits, as shown in Figure 5.

3.2.4 Prime Modulo Indexing

Prime modulo indexing (PRI) computes the index of a cache block address (A_B) using the following equation: $I = A_B \bmod p$, where p is the largest prime number that is equal to or smaller than the number of sets in the cache [11, 31]. The main advantage of PRI is that it is resistant to a wide range of memory strides, even including the aforementioned pathological cases for PLI [11].

However, given that PRI requires an integer division to compute the index for each cache access, it can introduce a significant indexing latency, potentially degrading the overall performance. While prior work proposed a fast implementation for PRI [11], it applied PRI only to the L2 cache because the additional indexing latency of PRI makes it infeasible for L1 caches. In addition, PRI is associated with from the *set fragmentation* problem [11], in which some of the sets in the cache are unutilized if the number of sets is not a prime number, which is a common case (e.g., a power of two). For instance, if a hardware cache consists of 32 sets, PRI only utilizes 31 sets, resulting in set fragmentation of 3.125% ($= \frac{1}{32} \times 100$).

3.3 Adaptive Cache Indexing

With regard to adaptive cache indexing (ADI) for GPGPUs, we adopt a technique referred to as ASCIB, that has been proposed to adjust the cache indexing bits dynamically to reduce conflict misses for CPU caches [32]. We choose ASCIB as the baseline design owing to its simplicity and applicability to L1 caches.

The ADI phase consists of three phases: victimization, selection, and idle phases (Figure 6). The *victimization* phase determines the victim bit from among the current indexing bits, which contributes the least to reducing set conflicts. For this purpose, ADI computes the *entropy* of each of the current indexing bits to quantify its variability. The entropy counter (C_E) of each indexing bit increments by 1, if the corresponding indexing bit value is 1. Assuming that the

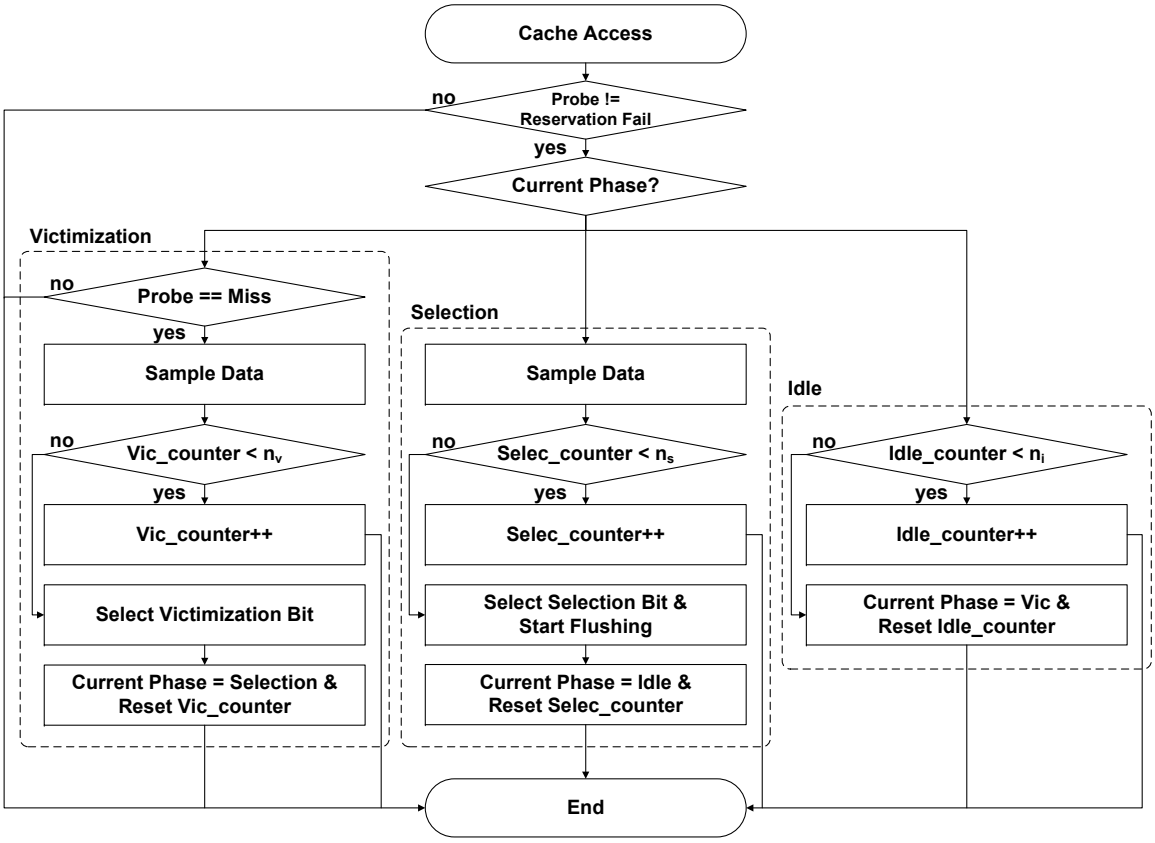


Figure 6: The overall execution flow of the adaptive cache indexing scheme

$\mathbf{B_2}B_1B_0$

0 0 1	entropy(B_0)=MIN(3,4-3)=1	corr(B_0,B_1)=MAX(3,4-3)=3
0 1 1	entropy(B_1)=MIN(2,4-2)=2	corr(B_0,$\mathbf{B_2}$)=MAX(3,4-3)=3
0 0 1	entropy($\mathbf{B_2}$)=MIN(0,4-0)=0	corr(B_1,$\mathbf{B_2}$)=MAX(2,4-2)=2
0 1 0		

Figure 7: An example of determining a victimized bit

total sample count is T, the entropy of each indexing bit is computed to be $MIN(C_E, T - C_E)$. In addition, ADI computes the *correlation* of every pair of the current indexing bits. The correlation counter (C_C) increments by 1 if the indexing bits in the corresponding pair have the same value. The correlation of each pair is computed to be $MAX(C_C, T - C_C)$.

ADI employs a metric called *usefulness* to compare the entropy of each of the indexing bits and the correlation of each pair of the indexing bits directly. With a total sample count of T, the usefulness of the entropy (i.e., E) of each of the indexing bits is computed to be E and the usefulness of the correlation (i.e., C) of each pair of the indexing bits is computed to be $T - C$. If the indexing bit with the lowest entropy has a lower usefulness value than the pair of the two indexing bits with the highest correlation, the indexing bit with the lowest entropy is selected as the victim bit. Otherwise, the indexing bit that has lower entropy between the two indexing

Candidates B_2B_1	Index B_0
0 0	1
1 0	1
0 1	0
0 1	1
1 1	0

When B_0=0		When B_0=1	
B_2	B_1	**B_2**	B_1
0	1	0	0
1	1	1	0
		0	1

When B_0=0	When B_0=1
$MP_0(B_1)$=2/1: 11	$MP_1(B_1)$=3/2: 001
$MP_0(\mathbf{B_2})$=2/2: 01	$MP_1(\mathbf{B_2})$=3/3: 010
$MRP(B_1)=(MP_0(B_1)+MP_1(B_1))/2$=7/4	
$MRP(\mathbf{B_2})=(MP_0(\mathbf{B_2})+MP_1(\mathbf{B_2}))/2$=1	

Figure 8: An example of selecting a new indexing bit

bits with the highest correlation is selected as the victim bit. Figure 7 shows an example of the determination of a victimized bit (i.e., B_2). In this work, the victimization phase period is configured as 1K L1 data cache misses.

The hardware overhead of the victimization phase is estimated as follows. The victimization phase requires $\frac{S\cdot(S+1)}{2}$ counters to collect the entropy and correlation data, where 2^S is the number of sets in the cache. With a period of 2^M misses for the victimization phase, each counter requires M bits. Therefore, $\frac{M\cdot S\cdot(S+1)}{2}$ bits are required for the counters. In addition, the victimization phase requires $\frac{S\cdot(S+1)}{2}$ M-bit adders and $\frac{S\cdot(S-1)}{2}$ XOR gates to compute the entropy and correlation [32,33]. For instance, with a 16KB L1 data cache, the hardware overhead required for the victimization phase includes 150 bits for the counters, 15 10-bit adders, and 10 XOR gates when $S = 5$ and $M = 10$.

The *selection* phase determines the new indexing bit that is expected to be most effective for reducing set conflicts. ADI quantifies the effectiveness of each bit using a metric called the *mean relative period* (MRP). For a sequence of memory addresses, the *mean period* (MP) of an address bit position is defined as the average length of consecutive zeros or ones. The MRP of the bit position is defined as the average of the MPs. More specifically, the sequence of memory addresses is clustered based on the values of the current indexing bits (excluding the victim bit). For each cluster of addresses, the MP of the bit position is computed. The MRP of the bit position is computed as the average of the MPs across all clusters of addresses. The new indexing bit is then determined as the bit with the lowest MRP among all non-indexing bits. This is done because it is expected to change most frequently relative to the current indexing bits. In this work, the selection phase period is configured as 1K L1 data cache accesses.

Figure 8 shows an example of the selection of new indexing bit. The bit sequences of B_2 are 01 ($B_0 = 0$) and 010 ($B_0 = 1$) when $B_0 = 0$ and $B_0 = 1$, respectively. The MP of B_2 when $B_0 = 0$ is computed to be 1 (i.e., $\frac{2}{2} = 1$) because the length of the bit sequence is 2 (i.e., 01) and because there are two segments of consecutive zeros or ones (i.e., 0 and 1). Similarly, the MP of B_2 when $B_0 = 1$ is computed to be 1 (i.e., $\frac{3}{3} = 1$) because the length of the bit sequence is 3 (i.e., 010) and because there are three segments of consecutive zeros or ones (i.e., 0, 1, and 0). The MRP of B_2 is then computed to be 1 (i.e., the average of the MPs). Finally, B_2 is selected as the new indexing bit because it has the lowest MRP.

If the new indexing bit differs from the victim bit, all of the cache lines in the state other than

the invalid state must be flushed to guarantee correctness. In line with earlier results [32,33], our experimental results show that the performance degradation due to cache flushing is insignificant, as the process is not performed too frequently.

The hardware overhead of the selection phase is estimated as follows. The selection phase requires a tag cache that consists of 2^{S-1} entries. With the N-bit memory address space and 2^B-byte cache blocks, each entry requires $(N - S - B + 1)$ bits in order to compute the MRP of the candidate bits. With a period of 2^M accesses for the selection phase, each candidate bit requires an M-bit counter. Therefore, the hardware overhead required for the tag cache is $(N-S-B+1)\cdot(2^{S-1}+M)$ bits. In addition, $(N-S-B+1)$ M-bit adders and $(N-S-B+1)$ XOR gates are required to compute the MRP [32,33]. Further, S $logN$-bit registers are required to store the current indexing bit information. For example, with a 16KB L1 data cache, the hardware overhead required for the selection phase includes 546 bits for the tag cache and the counters, 21 10-bit adders, and 21 XOR gates when $N = 32$, $S = 5$, $B = 7$, and $M = 10$.

Finally, during the *idle phase*, the system runs without performing any monitoring or adaptation activities during a predefined period (i.e., 4K accesses). Note that all phases are fully decoupled from the critical path for L1 cache accesses. For instance, computations of metrics such as the entropy, correlation, and MRP are don separatly from the critical path. This property makes ADI applicable to L1 caches (as well as the L2 cache).

To adopt ADI for massively-parallel GPGPU architectures, we extend the baseline ADI as follows. First, we design ADI to support multiple outstanding memory requests robustly. Upon a cache miss, a cache line is reserved for the requesting thread. The underlying warp scheduler then schedules another warp, which is ready to run with minimal overhead through fast hardware context switching to maximize the throughput. While the prior memory request is still being handled by the hardware components in the memory hierarchy (e.g., DRAM), it is possible for ADI to change the cache indexing bits. If the corresponding cache line is immediately invalidated during the cache flushing step, subtle correctness issues may arise when the pending data arrives, which was originally indexed based on the previous cache indexing bits.

To address such correctness issues, we introduce a new cache-line state called *reserved-doomed* (RD), which indicates that the corresponding cache line is reserved for a pending memory request that was issued before the cache indexing bits changed to the current ones but that has not been completed. When a cache line is in the RD state, it cannot be reserved for any subsequent memory request. When the pending data arrives, the corresponding cache line transitions from the RD state to the invalid state and becomes available for subsequent memory requests.

In our design, the physical location (i.e., the set index and the way number) of each cache line in the reserved or RD state is encoded in the header of the corresponding request/response packet to/from a lower memory component in the hierarchy. We expect the overhead of this design to be low. For instance, as for the 4-way 16KB L1 data cache with the block size of 128 bytes, the extra bits required in the header of each memory request or response packet amount to 7 bits (i.e., 5 bits for the set index and 2 bits for the way number). Because the memory request

Time	Core	L1D Response	L1D Line State	NOC
0	ld A	Miss	I → R	Send request for A
1			R → RD (flush)	
2	ld B	Reservation Fail	RD	
3			RD → I	Receive A
4	ld B	Miss	I → R	Send request for B

Figure 9: An example of cache-line state transitions

and response packets already include the 8-byte control data (e.g., the address and SIMT core ID) [16], we believe that adding an extra 7 bits to encode the physical location of the reserved cache line incurs low overhead.

Figure 9 shows an example of how the state of a cache line changes when memory operations are performed on a system in which ADI is applied to its L1 data cache. At time 0, the SIMT core attempts to load data `A`. Because the request incurs an L1 data cache miss, the state of the corresponding cache line in the L1 data cache is changed from the invalid to the reserved state and the request is sent through the interconnection network. At time 1, the state of the cache line is changed to the reserved-doomed (RD) state due to the cache flushing triggered by ADI.[3] At time 2, the core attempts to load data `B`, which happens to be mapped to the same cache line as data `A`. Because the corresponding cache line is in the RD state, the request for data `B` cannot proceed due to the reservation fail. At time 3, data `A` is received from the interconnection network and the state of the cache line is changed to the invalid state. Finally, at time 4, the core retries to load the data `B`, incurring an L1 data cache miss.

Second, because the baseline ADI only supports a single core, we extend it to support multiple SIMT cores and L2 cache banks. To this end, we duplicate the hardware logic required for ADI across all the SIMT cores and L2 cache banks. With this design approach, ADI allows for each private L1 data cache and each L2 cache bank to adapt in a fully distributed manner without requiring any centralized hardware structure. As analyzed above and also quantified earlier [32] (i.e., the area overhead $< 2\%$), the area overhead required for ADI is expected to be low.

Third, for the L1 data cache, ADI filters out the addresses for the writes to global memory during the victimization and selection phases. This is done because most modern GPGPU architectures typically employ a policy with write evict (on write hits) and write no-allocate (on write misses) for global memory accesses without providing cache coherence across private L1 data caches.

[3]Note that cache flushing is typically performed in multiple cycles. To ensure correctness, the cache controller rejects any incoming memory request from the SIMT core until cache flushing is complete. For brevity, in this example, we assume that cache flushing is performed in a single cycle.

Table 1: Architectural parameters of the simulated system

Parameter	Value
SIMT Core	Core count: 16, SIMT width: 32, pipeline depth: 5, frequency: 700MHz
Per-core resource	Number of registers: 32768, scratchpad: 48KB, MSHRs: 32, warps: 48, threads: 1536
Schedulers	Warp scheduler: Greedy-Then-Oldest (GTO), CTA scheduler: round-robin
L1 data cache	Capacity: 16KB/core, line size: 128B, associativity: 4, coalescing: enabled
Interconnect	Frequency: 700MHz, channel width: 32
L2 cache	Capacity: 64KB/bank, number of banks: 12, line size: 128B, associativity: 8
DRAM	Frequency: 924MHz, scheduler: FR-FCFS, number of MCs: 6, channel BW: 4B/cycle

3.4 Evaluation

This section provides a quantitative evaluation of advanced cache indexing (ACI) for GPGPU computing. Specifically, we aim to investigate the following – the effectiveness of ACI for L1 data and L2 caches and the sensitivity of ACI to architectural parameters such as the indexing latency, cache capacity, and associativity.

3.4.1 Methodology

We implemented the ACI schemes in the GPGPU-Sim simulator (version 3.2.2) [16]. We use architectural parameters similar those defined in the configuration file in the GTX480 directory (Table 1). We investigate the performance and energy efficiency of the following cache indexing schemes – conventional (CVI), bitwise XOR (BXI), reverse-engineered bitwise XOR (RXI), polynomial modulus (PLI), prime modulo (PRI), and adaptive indexing (ADI) schemes. Regarding ADI, the periods of the victimization, selection, and idle phases are set to 1K misses, 1K accesses, and 4K accesses to the L1 data and L2 caches, respectively. Note that the L1 data and L2 caches with the conventional and ACI schemes are configured with the same capacity and associativity. We use the Greedy-Then-Oldest (GTO) warp scheduler [3]. To quantify the energy efficiency of the ACI schemes, we use GPUWattch [34].

Table 2 summarizes all of the benchmarks that we use to investigate the effectiveness of the ACI schemes. The benchmarks are selected from the commonly used GPGPU benchmark suites proposed in [28,35–38]. The benchmarks exhibit widely different memory access characteristics. Inspired by the classification presented in [6], we classify the benchmarks into five categories

Table 2: Benchmarks

Category	Name	Description
Streaming	HS	HotSpot [35]
	NW	Needleman-Wunsch [35]
	BLK	Black Scholes [36]
	CONV	Convolution [36]
	FWT	Fast Walsh Transform [36]
Conflicting	2DC	2D Convolution [28]
	2MM	2 Matrix Multiplications [28]
	SRAD	Speckle Reducing Anisotropic Diffusion [35]
	SC	Streamcluster [35]
Thrashing	BFS	Breadth-First Search [35]
	KM	Kmeans [35]
	II	Inverted Index [37]
	SPMV	Sparse Matrix-Vector Multiplication [38]
Conflicting & Thrashing	ATAX	Matrix Transpose and Vector Multiplication [28]
	GSM	Scalar, Vector and Matrix Multiplication [28]
	SYRK	Symmetric Rank-K Operations [28]
Friendly	BP	Back Propagation [35]
	BT	B+ Tree [35]
	NN	Nearest Neighbor [35]
	OP	Monte Carlo Option Pricing [36]

based on their memory access characteristics – streaming, conflicting, thrashing, conflicting and thrashing, and cache friendly benchmarks.

3.4.2 Effectiveness of Advanced Cache Indexing for the L1 Data Cache

First, we investigate the performance and energy efficiency of the ACI schemes. Figures 10 and 11 show the overall performance (i.e., instruction per cycle (IPC)) and energy consumption normalized to the conventional cache indexing scheme when the ACI schemes are applied to the L1 data cache. To evaluate their potential for improving the performance and energy efficiency, we assume that the ACI schemes do not incur any extra overhead (e.g., indexing latency), which may be rather optimistic for the sophisticated cache indexing schemes such as PLI and PRI. Section 3.4.4 quantifies the performance sensitivity of PLI to the indexing latency.

Figure 10 demonstrates that the ACI schemes significantly improve the performance of some benchmarks, especially the conflicting and conflicting and thrashing (C+T) benchmarks. Specifically, BXI, RXI, PLI, PRI, and ADI provide corresponding performance improvement of 40.2%, 39.8%, 47.0%, 45.8%, and 42.3% over the conventional indexing scheme on average (i.e., geometric mean). In addition, the ACI schemes incur little or no performance degradation across all of the evaluated benchmarks.

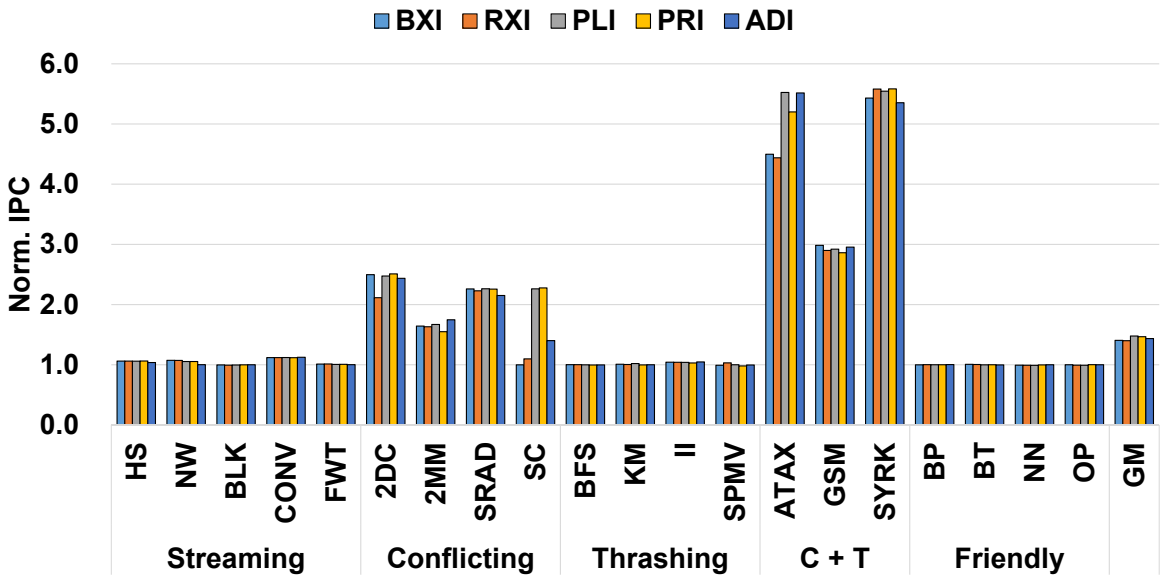


Figure 10: Overall performance results of the ACI schemes applied to the L1 data cache

Figure 11 shows that the ACI schemes significantly reduce the energy consumption, especially the conflicting and conflicting and thrashing (C+T) benchmarks. Specifically, BXI, RXI, PLI, PRI, and ADI provide corresponding energy reductions of 10.7%, 11.0%, 13.2%, 12.7%, and, 12.1% over the conventional indexing scheme on average (i.e., geometric mean). Similarly to the performance result trend, the ACI schemes incur little or no increase in their energy consumption levels across all the evaluated benchmarks.

To gain deeper insight into the performance and energy results, we provide detailed cycle and energy breakdowns of a subset of the evaluated benchmarks. To maintain variety yet conciseness, we investigate the seven benchmarks among all the evaluated benchmarks by selecting at least one benchmark from each category which exhibits different performance and energy trends when the ACI schemes are applied.

Figure 12 shows the execution cycle breakdown of the seven benchmarks. For each benchmark, we run it with 6 different indexing configurations for the L1 data cache. Each bar is normalized to the baseline version in which the conventional indexing scheme is used for the L1 data and L2 caches. Each bar consists of multiple segments, each indicating busy cycles, idle cycles due to a load imbalance across SIMT cores (Idle Core), idle cycles spent when no warp is ready to execute (Idle Warp), cycles spent for ALU (PL ALU), LDST (PL LDST), and both (PL Both) pipeline stalls, and cycles stalled at the scoreboard, waiting for the data produced by ALU (SB ALU), LDST (SB LDST), and both (SB Both) instructions.

Figure 12 shows that the ACI schemes significantly improve `2DC`, `ATAX`, and `SYRK` performance outcomes. For these benchmarks, the ACI schemes effectively reduce L1 data cache misses (Figure 13(a)) and reservation fails (Figure 13(b)), which eventually reduces the LDST stall cycles (i.e., the “PL LDST" segment in Figure 12). In particular, regarding `ATAX`, ADI outperforms all

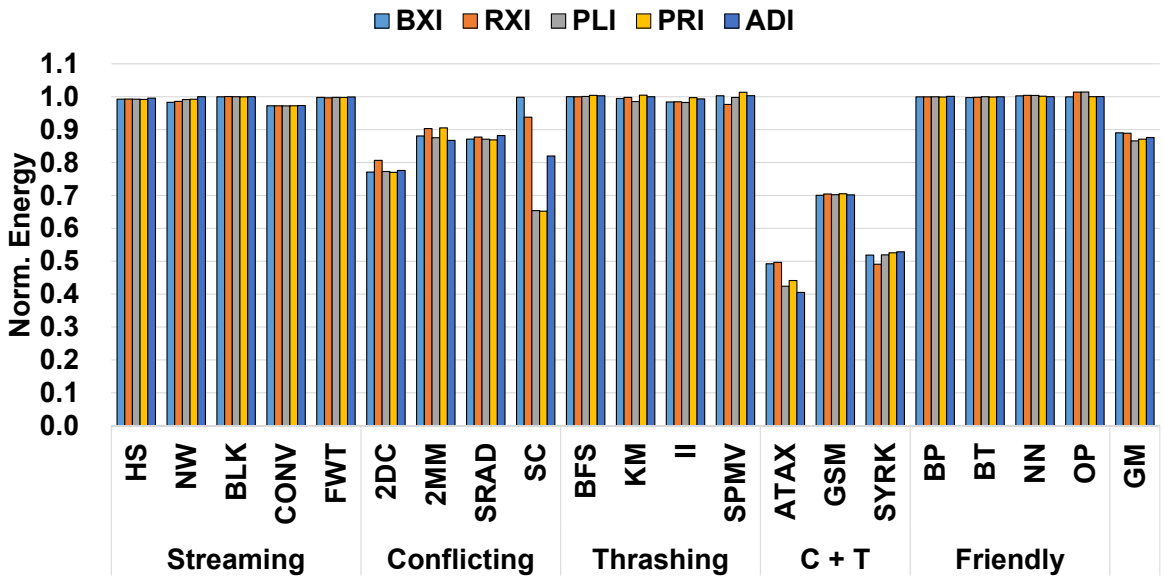


Figure 11: Overall energy results of the ACI schemes applied to the L1 data cache

static cache indexing schemes by successfully identifying and utilizing the important bits as the indexing bits based on the runtime information (Section 3.1).

Some of the ACI schemes show different performance trends than the others. For `NW`, ADI is ineffective compared to the other schemes. This is mainly due to the short kernel execution time of `NW`, eventually providing insufficient adaptation opportunities for ADI.

For `SC`, BXI, RXI, and ADI are outperformed by PLI and PRI. Some of the high bits in the memory address are effective for reducing the cache contention of `SC`. However, BXI, RXI, and ADI are designed mostly to utilize the lower bits for constructing the indexing bits to provide higher performance across a wide range of applications (BXI and RXI) and maintain low hardware complexity (ADI) [32]. Therefore, BXI, RXI, and ADI show suboptimal performance for `SC`. In contrast, PLI and PRI achieve higher performance by effectively utilizing the most significant bits.

For `2DC`, RXI is significantly outperformed by BXI. This occurs mainly because some of the higher bits employed by RXI to construct the indexing bits are less effective for mitigating the intra-warp interference incurred in `2DC`. Figures 13(a) and 13(b) also show that RXI incurs significantly more L1 data cache misses and reservation fails than BXI.

Finally, the performance impact of the ACI schemes is negligible for `BFS` and `BT`. This is mainly because they are highly hand optimized schemes to facilitate the coalescing of the memory addresses requested by the threads in the same warp, limiting the effect of the ACI schemes. For these benchmarks, Figure 13 shows that the L1 data cache misses and reservation fails remain rather unaffected when the ACI schemes are applied to the L1 data cache.

We now investigate the energy efficiency of the ACI schemes. Figure 14 shows the energy consumption normalized to the conventional indexing scheme. The L1 data cache, NOC, L2,

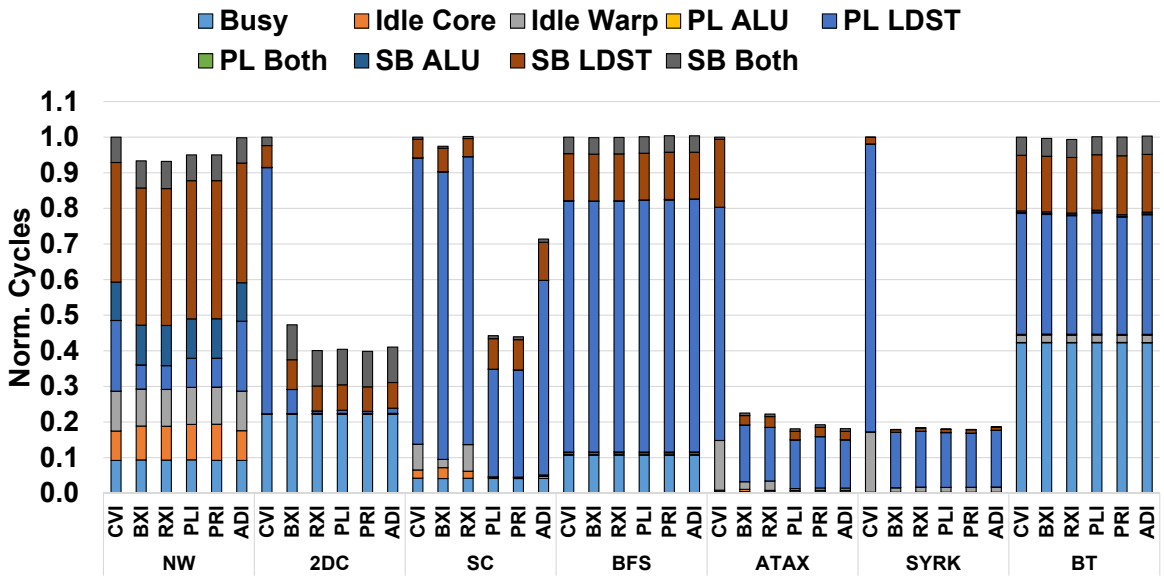
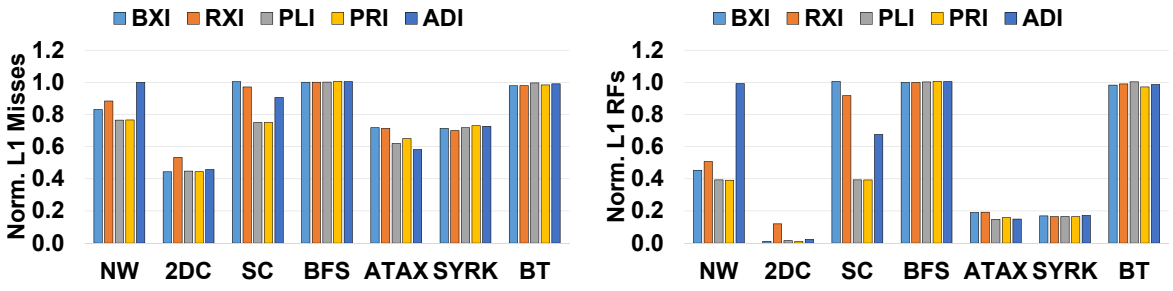


Figure 12: Execution cycle breakdown of the conventional and ACI schemes applied to the L1 data cache



(a) Misses

(b) Reservation fails

Figure 13: L1 data cache misses and reservation fails of the conventional and ACI schemes applied to the L1 data cache

DRAM+MC, idle core, and others (mainly related to SIMT cores) segments indicate the energy consumed by the corresponding hardware components. Figure 14 shows that the ACI schemes significantly improve the energy efficiency for a subset of the evaluated benchmarks (i.e., `2DC`, `SC`, `ATAX`, and `SYRK`), demonstrating the potential of ACI schemes to realize high performance and energy efficient GPGPU computing. For these benchmarks, we observe that the energy reduction is mainly achieved from the reduction in the non-memory hardware components (i.e., the idle core and others). Similar to the performance results, the energy consumption of `BFS` and `BT` remains unaffected with the ACI schemes.

To gain a deeper understanding of the energy reduction realized with ACI schemes, Figure 15 shows the power consumption normalized to the conventional indexing scheme. For the benchmarks that result in an energy reduction with the ACI schemes, we observe that power consumed

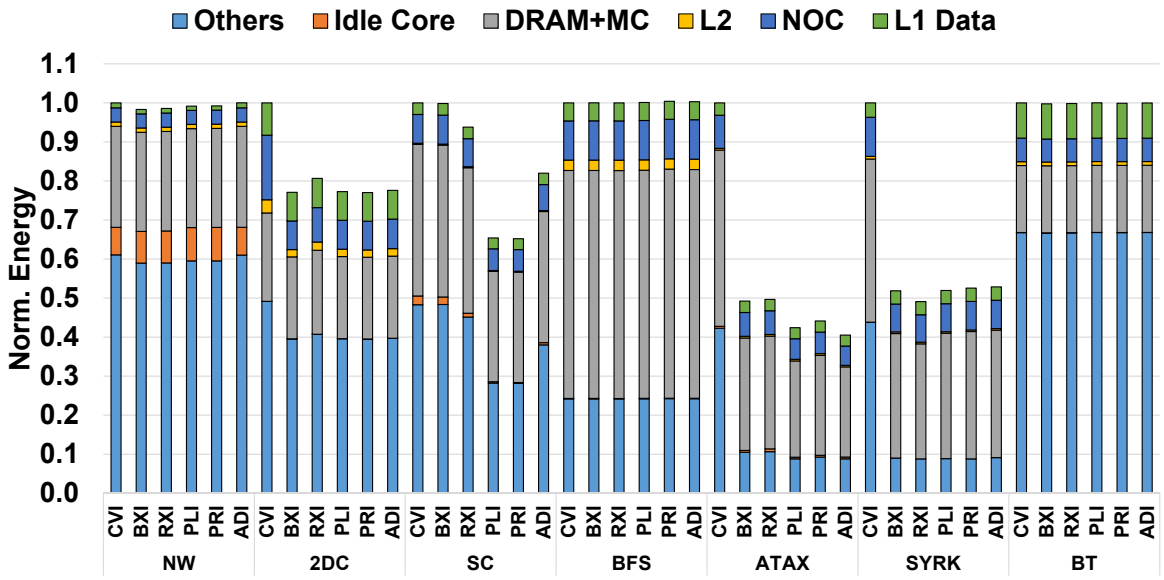


Figure 14: Energy consumption breakdown of the conventional and ACI schemes applied to the L1 data cache

by the hardware components (e.g., DRAM and memory controllers) in the memory hierarchy increases. Note that, with the ACI schemes, reservation fails in the L1 data caches are significantly reduced. This allows for more memory requests per unit time to be performed through the hardware components in the memory hierarchy, increasing their dynamic power consumption. However, because the overall memory traffic is reduced with fewer L1 data cache misses (Figure 13(a)), ACI marginally reduces the energy consumption of the hardware components in the memory hierarchy.

Figure 15 shows that the power consumption of the non-memory hierarchy components (i.e., idle cores and others) is less significantly affected than the hardware components in the memory hierarchy with the ACI schemes. This occurs because the non-memory hierarchy components continues to consume dynamic power even for instructions that have been issued but have failed to execute due to events such as reservation fails. In addition, the "others" segment includes the static power consumption, which is not affected by the use of the ACI schemes. As such, combined with the reduced execution cycles, the energy reduction stemming from the non-memory hierarchy components accounts for a significant portion of the total energy reduction for the aforementioned benchmarks with the ACI schemes.

3.4.3 Effectiveness of Advanced Cache Indexing for the L2 Cache

This section investigates the effectiveness of ACI for the L2 cache.[4] For each benchmark, we run it with 8 different cache indexing configurations. Each configuration is labeled with two

[4]Because the RXI scheme has been reverse-engineered only for the L1 data cache but not for the L2 cache [29], we are unable to investigate the effectiveness of RXI with regards to the L2 cache.

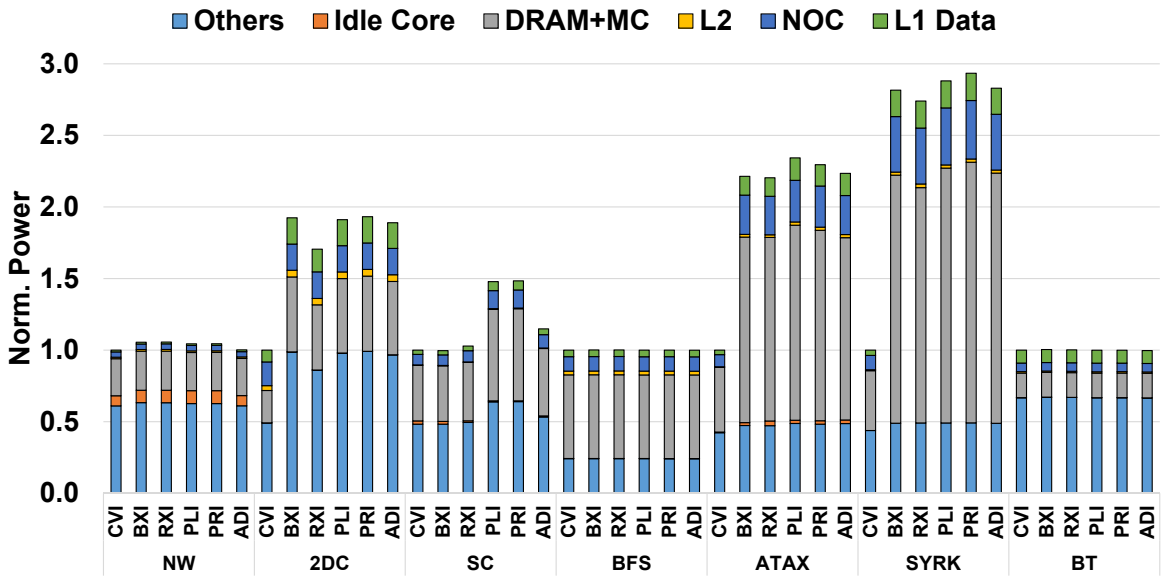


Figure 15: Power consumption breakdown of the conventional and ACI schemes applied to the L1 data cache

letters. The first and second letters denote the cache indexing scheme used for the L1 data and L2 caches, respectively. Each letter indicates one of the following cache indexing schemes – conventional (C), bitwise XOR (B), polynomial modulus (L), prime modulo (R), and adaptive (A) indexing schemes. For example, the LL configuration indicates that PLI is used for the L1 data and L2 caches. For another example, the AC configuration indicates that the adaptive and conventional indexing schemes are used for the L1 data and L2 caches, respectively. Using these configurations, we quantify the additional performance and energy efficiency gains of the ACI schemes applied to the L2 cache.

For all 20 evaluated benchmarks listed in Table 7, BXI, PLI, PRI, and ADI improve the performance by 57.1%, 69.9%, 65.7%, and 59.8% and reduce the energy consumption by 28.0%, 32.3%, 29.3%, and 27.9%, respectively, over the conventional indexing scheme. To keep the discussion concise and focused, we discuss the experimental results of the four benchmarks (i.e., `2DC`, `SC`, `BFS`, and `ATAX`), each of which exhibits different performance and energy efficiency trends with the ACI schemes applied to the L2 cache.

Figure 16 shows the execution cycle breakdown normalized to the results of the conventional indexing scheme applied to the L1 data and L2 caches. To quantify the additional performance gain of the ACI schemes applied to the L2 cache, we report the results with the schemes only applied to the L1 data cache and both the L1 data and L2 caches. For `2DC`, the ACI schemes applied to the L2 cache have an insignificant performance effect because the memory controller and DRAM form the major performance bottleneck due to bandwidth saturation.

In contrast, the ACI schemes provide a significant additional performance gain for `SC`, `BFS`, and `ATAX` when they are applied to both the L1 data and L2 caches, mainly because these

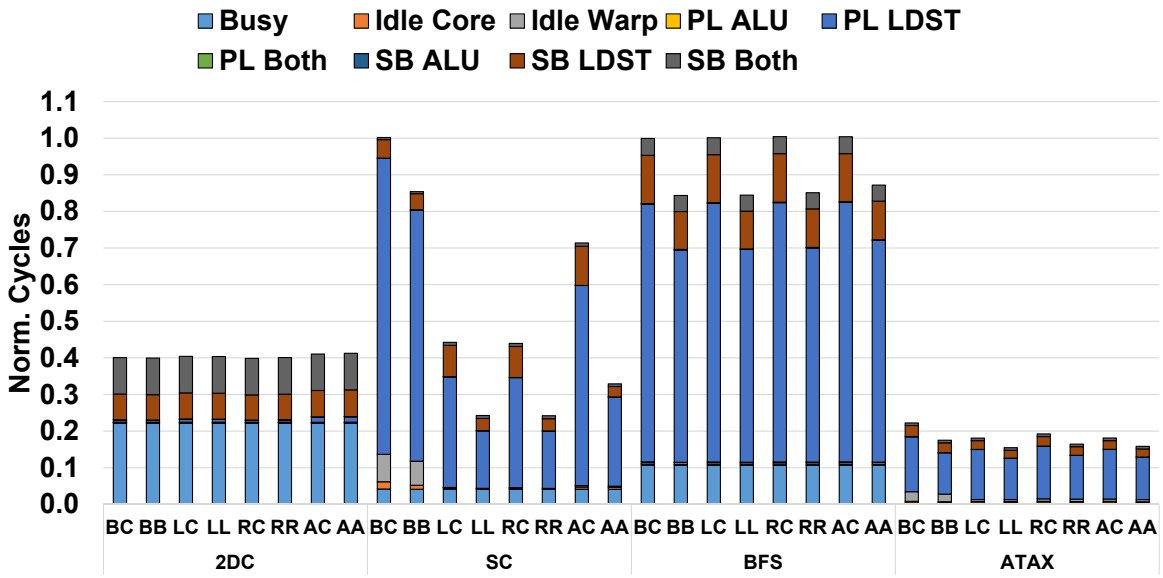
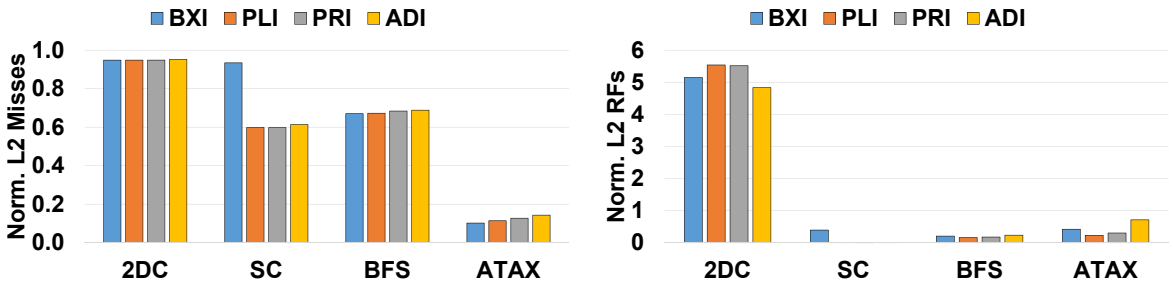


Figure 16: Execution cycle breakdown of the conventional and ACI schemes applied to the L1 data and L2 caches



(a) Misses

(b) Reservation fails

Figure 17: L2 data cache misses and reservation fails of the conventional and ACI schemes applied to the L1 data and L2 caches

schemes effectively reduce the number of L2 reservation fails (Figure 17(b)), eventually leading to a reduction in the LDST pipeline (i.e., the PL LDST segment in Figure 16) and fewer scoreboard stalled (i.e., the SB LDST segment) cycles.[5]

BXI and ADI provide smaller performance gains for `SC` than for the other ACI schemes because they do not employ most significant bits, which are highly effective for mitigating the cache contention issue, for indexing[6] For instance, Figure 17(a) shows that BXI incurs

[5]For `2DC`, the number of reservation fails in the L2 cache greatly increase with the ACI schemes because significantly more memory requests per cycle are sent to the L2 cache due to the reduced reservation fails in the L1 data cache enhanced with the ACI schemes.

[6]As in case of the L1 data cache, BXI and ADI applied to the L2 cache are designed to utilize the lower bits of a memory address when constructing the indexing bits to achieve higher performance across a wide range of applications (BXI) and to maintain low hardware complexity (ADI).

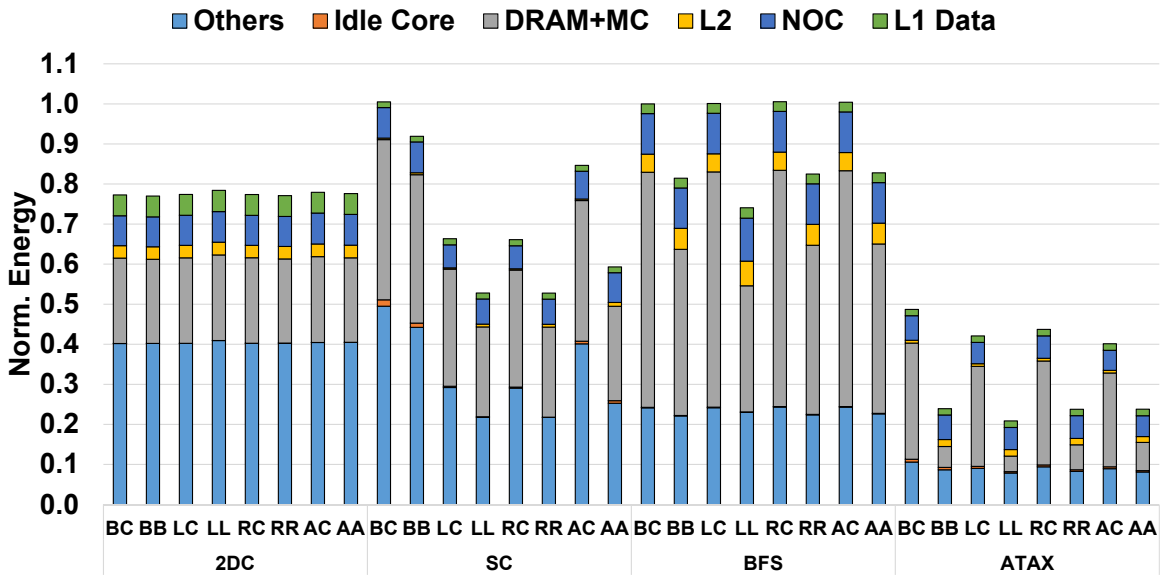


Figure 18: Energy consumption breakdown of the conventional and ACI schemes applied to the L1 data and L2 caches

significantly more L2 cache misses than the other ACI schemes. In addition, it is interesting that while the ACI schemes have insignificant performance effects on the L1 cache for `BFS`, they significantly improve the performance when they are applied to the L2 cache. This clearly demonstrates the importance of holistically optimizing the hardware caches in the GPGPU memory hierarchy.

We now investigate the energy efficiency of the ACI schemes applied to the L2 cache. Figure 18 shows the energy consumption breakdown normalized to the results of the conventional indexing scheme applied to the L1 data and L2 caches. For `2DC`, the ACI schemes have insignificant effects on energy consumption because the major performance bottlenecks are the memory controller and DRAM.

In contrast, the ACI schemes significantly reduce the energy consumption of `SC`, `BFS`, and `ATAX` when they are applied to both the L1 data and L2 caches. In a closer analysis of the energy consumption results, Figure 19 shows the power consumption breakdown normalized to the results of the conventional indexing scheme applied to the L1 data and L2 caches. The ACI schemes applied to the L2 cache increase the power consumption of `SC` significantly, mainly due to the increased power consumption of the memory hierarchy components (e.g., NOC, DRAM+MC) because more memory requests per cycle can be transmitted through the memory hierarchy by effectively mitigating the L2 cache contention issue using the ACI schemes. However, the performance improvement of the ACI schemes effectively cancels out the increased power consumption, significantly reducing the energy consumption of `SC`.

Interestingly, the ACI schemes applied to the L2 cache reduce both the energy and power consumption outcomes for `BFS` and `ATAX`. The ACI schemes significantly reduce L2 cache misses

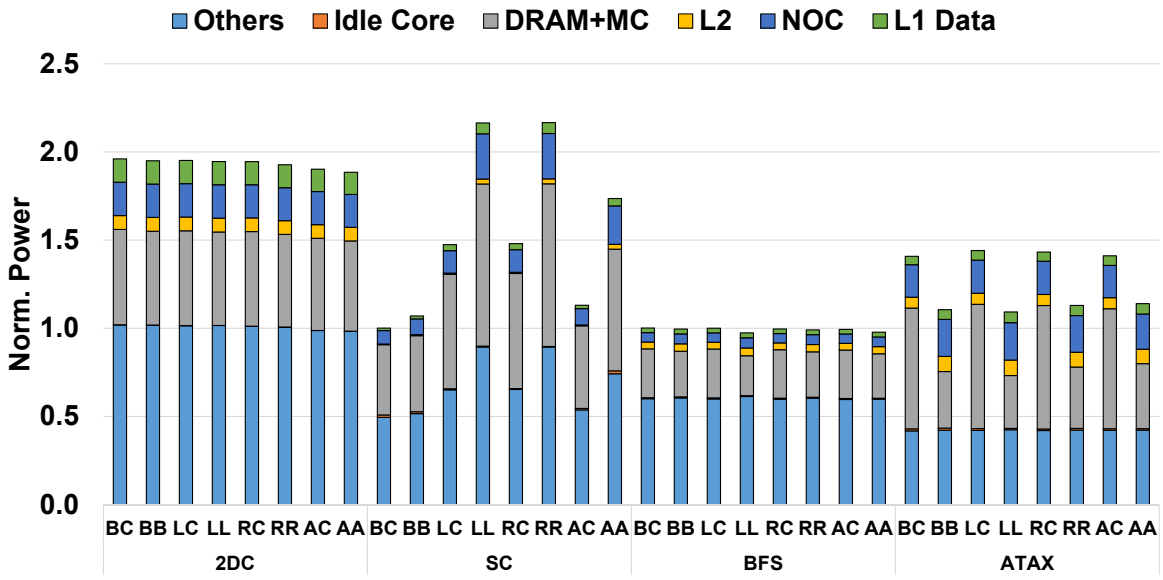


Figure 19: Power consumption breakdown of the conventional and ACI schemes applied to the L1 data and L2 caches

with these benchmarks as well. In other words, the L2 cache effectively filter out the memory requests that would be sent to the memory controller and DRAM when using the conventional indexing scheme, reducing the power consumption by these components. Combined with the reduced power consumption and fewer execution cycles, the ACI schemes significantly reduce the overall energy consumption of `BFS` and `ATAX`.

3.4.4 Sensitivity of Advanced Cache Indexing to Architectural Parameters

This section investigates the performance sensitivity of the ACI schemes to key architectural parameters, in this case the indexing latency, cache associativity, and capacity. To compile the performance sensitivity results, we apply the advanced indexing schemes to the L1 data cache and the conventional indexing scheme to the L2 cache.

Given the rather complicated hardware logic of some of the ACI schemes (e.g., PLI) located along the critical path, they can introduce additional indexing latency [30], especially as the SIMT core frequency aggressively scales in the future generation of GPGPU architectures. To quantify the performance effect of the additional indexing latency, we compare the performance of PLI by sweeping the additional indexing latency from 0 to 4. We investigate the performance sensitivity of PLI to indexing latency because (1) the hardware logic of PLIthat computes the indexing bits is in the critical path, and (2) this scheme shows high performance and energy efficiency across a variety of the evaluated benchmarks.

On average, the performance impact of the additional latency is rather insignificant. Specifically, the average performance degradation rates across all the 20 benchmarks listed in Table 7 are 0.9%, 1.3%, 2.8%, and 5.7% when the additional indexing latency is set to 1, 2, 3, and 4

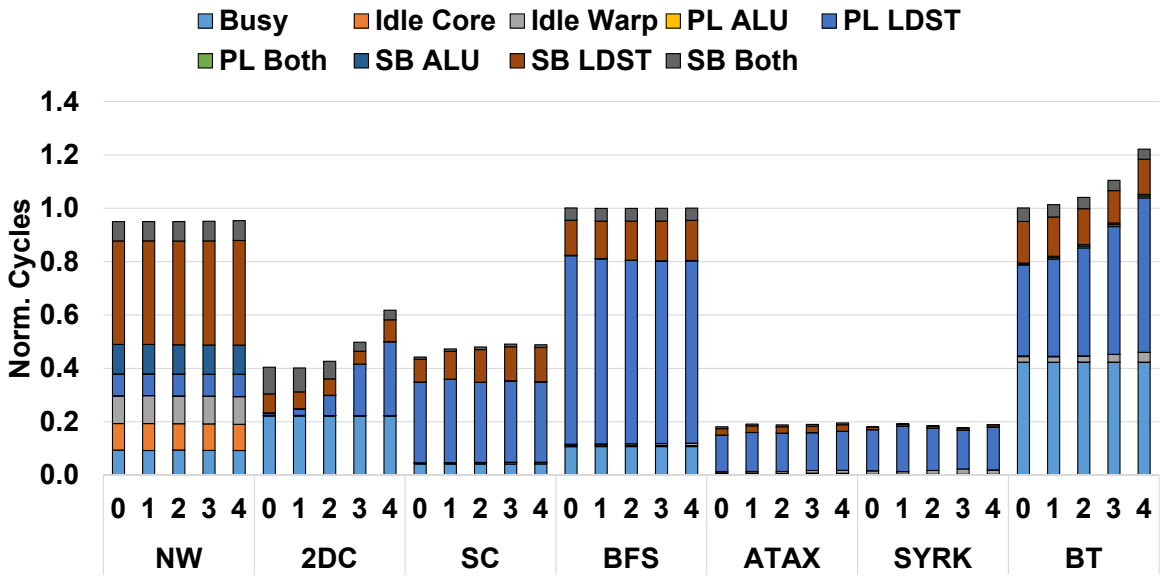


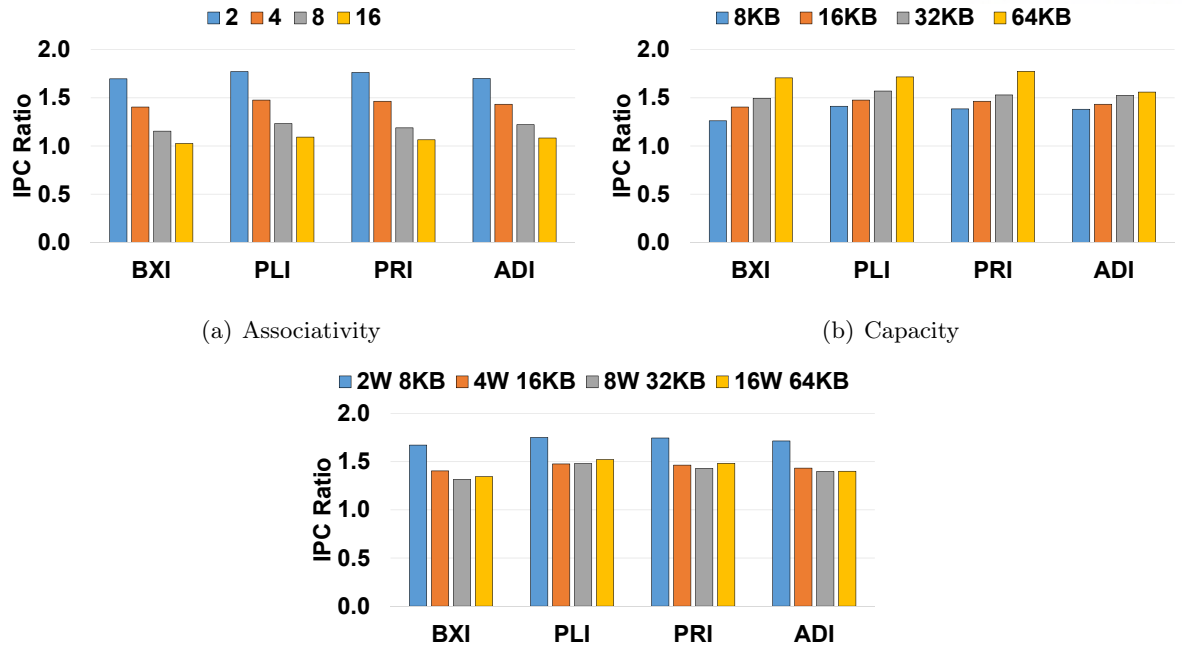
Figure 20: Sensitivity to the indexing latency

cycles, respectively. This occurs mainly because some of the benchmarks are already associated with a number of reservation fails per access even when the L1 data cache is augmented with PLI. Meanwhile, other benchmarks show decreased performance outcomes, as the additional indexing latency directly affects the performance of the benchmarks. Therefore, it is important to apply PLI judiciously depending on the application characteristics and additional indexing latency.

Figure 20 shows the execution cycles (normalized to the conventional indexing scheme) of PLI by sweeping the additional indexing latency from 0 to 4 cycles. For the benchmarks (i.e., `NW`, `SC`, `BFS`, `ATAX`, and `BT`) resulting in a large number of reservation fails per L1 data cache access (e.g., 18.5 for `NW`), the additional indexing latency causes little performance degradation (0.6% for `NW` with 4-cycle additional indexing latency). In contrast, for the other benchmarks (i.e., `2DC` and `BT`) that cause a relatively small number of reservation fails per L1 data cache access (e.g., 0.11 for `2DC`), the additional indexing latency incurs significant performance degradation (45% for `2DC` with 4-cycle indexing latency). In summary, the performance sensitivity results of PLI to indexing latency indicate that sophisticated ACI schemes such as PLI can be effective for high performance and energy efficient GPGPU computing depending on the additional indexing latency and the characteristics of the application.

We now investigate the performance sensitivity of the ACI schemes to L1 data cache associativity. Intuitively, we quantify how the performance gain of the ACI schemes changes as the baseline L1 data cache is augmented with high associativity. Figure 21(a) shows the IPC ratio[7] of the ACI schemes relative to the conventional indexing scheme as the L1 data cache

[7]For each cache indexing scheme, Figures 21(a), 21(b), and 21(c) report the geometric mean of the IPCs collected from all 20 benchmarks presented in Table 7.



(a) Associativity

(b) Capacity

(c) Associativity and capacity

Figure 21: Sensitivity to the L1 data cache associativity and capacity

associativity scales from 2 to 16 with the capacity to 16KB.

We observe that the performance gain of the ACI schemes over the conventional indexing scheme decreases as the L1 data cache associativity increases, as the baseline cache with conventional indexing becomes less vulnerable to cache contention with higher associativity. Nevertheless, the ACI schemes continue to provide performance gains even over the baseline cache augmented with high associativity (e.g., 16), which is considered to be highly challenging to implement without sacrificing performance, especially for L1 caches. In addition, the experimental results suggest that it is valuable to investigate techniques that can effectively increase the degree of cache associativity with little or no performance overheads [39–41].

We then investigate the performance sensitivity of the ACI schemes to the L1 data cache capacity. Figure 21(b) shows the IPC ratio of the ACI schemes to the conventional cache indexing scheme as the L1 data capacity scales from 8KB to 64KB while the associativity is held constant at 4. The performance gain of the ACI schemes increases with a larger cache capacity. Because the conventional indexing scheme continues to be plagued with high cache contention, even with a larger number of sets (from the larger cache) and with the workloads that exhibit pathological access patterns.

In contrast, the ACI schemes effectively mitigate the potential cache contention due to pathological memory accesses and fully benefit from the larger cache capacity, providing increasing performance gains over the conventional indexing scheme. The performance gain of ADI scales less effectively than the other ACI schemes because the performance overhead caused by flushing increases with a larger cache capacity.

Finally, we assess the performance sensitivity of the ACI schemes when both the associativity and capacity of the L1 data cache scale. Figure 21(c) shows the IPC ratio of the ACI schemes to the conventional indexing scheme with the 2-way 8KB, 4-way 16KB, 8-way 32KB, and 16-way 64KB L1 data caches, respectively. As discussed above, the performance gain of the ACI schemes tends to decrease with higher associativity and increase with larger capacity. Because the net performance effects of increasing the associativity and capacity cancel out, we observe rather small differences in the performance gains across different associativity and capacity configurations.

In summary, the performance sensitivity results in Figure 21 clearly demonstrate that the ACI schemes are promising in the sense that they continue to provide significant performance gains over the baseline cache even when they are enhanced with higher associativity and a larger capacity.

3.4.5 Discussion

Our experimental results show that PLI achieves the highest performance and best energy efficiency among the ACI schemes when applied to the L1 data cache or to both the L1 data and L2 caches. Considering the higher performance, better energy efficiency, and lower hardware complexity of PLI compared to PRI, PLI appears to be more effective than PRI for GPGPU computing. In particular, none of the 20 benchmarks evaluated here seems to exhibit pathological performance behaviors for PLI [11]. However, as shown in Section 3.4.4, PLI might suffer from performance degradation from additional indexing latency.

ADI shows slightly lower performance and energy efficiency outcomes compared to those of PLI and PRI, as some of these benchmarks do not benefit from ADI due to their short kernel execution cycles and because ADI only utilizes a subset of the bits in the memory address to keep the hardware complexity low. However, ADI has less complicated hardware logic in the critical path compared to other ACI schemes (e.g., PLI). Therefore, if additional indexing latency of PLI and PRI is determined to be too high, ADI will offers bettter performance and greater energy efficiency.

Finally, BXI and RXI show lower performance and energy efficiency outcomes than the other ACI schemes because they use simpler hash functions than PLI and PRI and only utilize a subset of the memory-address bits. Especially, BXI perform worse than RXI depite having same hardware complexity. This is because BXI utilizes bits ineffective for mitigating cache contentions (i.e., lower $2N$ bits) while RXI has been optimized for GPGPUs and utilizes higher bits.

IV Improving the Performance and Energy Efficiency of GPGPU Computing through Integrated Adaptive Cache Management

4.1 Motivation

The need for adaptation: GPGPU cache performance outcomes are mainly determined by how many threads share the cache, how evenly the cache indexing function distributes memory requests across the cache sets, and what memory requests are worth being served by the cache or bypassed. The cache management techniques that effectively handle each of these considerations include warp limiting, cache indexing, and cache bypassing.

Given the disparate architectural characteristics of GPGPU workloads, static cache management is highly likely to fail to achieve the best possible performance and energy efficiency for various GPGPU workloads. For instance, recent studies show that adaptive techniques are more effective than static methods for cache indexing [12], warp limiting [3], and cache bypassing [4].

The need for integration: Given the complex interference patterns among warps (and threads), there is no single cache management technique that is effective across all the performance pathological scenarios. For example, while adaptive warp limiting (AWL) is an effective and versatile technique, it cannot address the performance degradation caused by intra-warp interference because it cannot control threads within the same warp [5]. Therefore, multiple cache management techniques must be exploited in a tightly integrated manner to effectively address a wide range of performance pathologies more effectively (e.g., intra and inter-warp interferences) in GPGPU computing.

Further, integrating adaptive cache management techniques can provide more opportunities to achieve even higher performance and greater energy efficiency by facilitating constructive interactions between them. For instance, with a judiciously controlled warp count, the performance gain of adaptive cache indexing (ADI) can be significantly increased [12]. Similarly, AWL may dynamically increase the warp count if ADI effectively mitigates the contention among the concurrent warps with high-quality indexing, resulting in improved performance. Therefore, considering the potential of integrated adaptive cache management, it is important to create a fully integrated architectural framework and thoroughly quantify its effectiveness based on the integrated architecture.

4.2 The IACM Architecture

4.2.1 Overall Architecture

This section discusses the design and implementation of integrated adaptive cache management (IACM) for high-performance and energy-efficient GPGPU computing. Figure 22 shows the baseline GPGPU architecture that we extend to implement integrated adaptive cache management (IACM). Because the baseline GPGPU architecture is generic and similar to widely-used

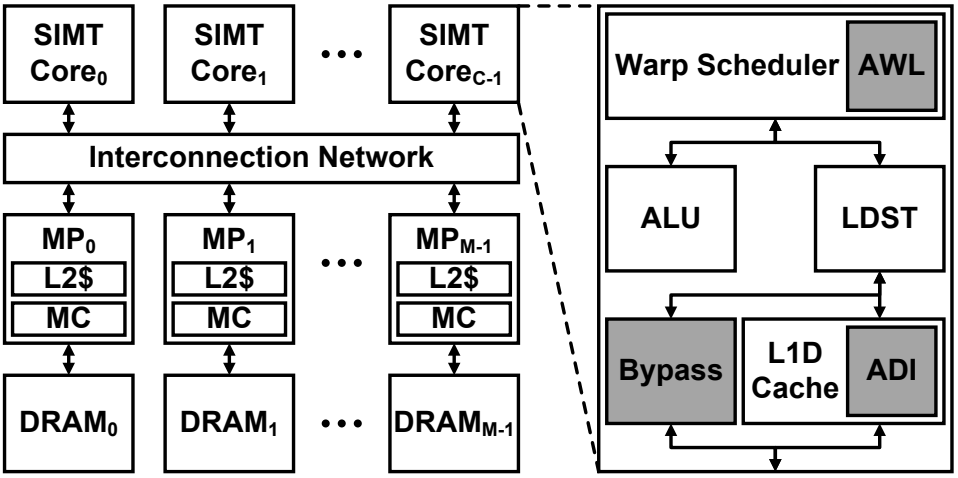


Figure 22: The baseline GPGPU architecture augmented with IACM

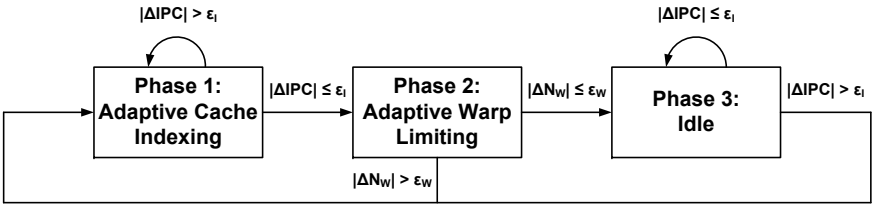
commercial architectures such as those of NVIDIA [1] and AMD [2], we believe that IACM can be applied to a wide range of GPGPU architectures. Note that the hardware components added for IACM (i.e., AWL, ADI, and Bypass) are shown in grey color, which will be discussed later in this section.

Figure 23 shows the overall execution flow of IACM, which consists of three execution phases – the adaptive cache indexing (ADI), the adaptive warp limiting (AWL), and the idle phases. The ADI phase dynamically determines the cache indexing bits that can reduce cache thrashing and contention based on the runtime information of the GPGPU workload. The AWL phase dynamically controls the number of active warps to improve performance and energy efficiency. The idle phase monitors the behavior of GPGPU workload without performing any adaptation and triggers the entire adaptation process again if it detects a significant phase change of the GPGPU workload.

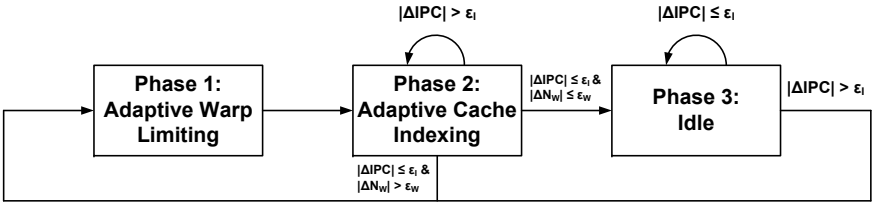
We present three IACM designs, each of which composes the adaptive cache management techniques in a different manner. The ADI-AWL version of IACM shown in Figure 23(a) performs ADI before AWL, whereas the AWL-ADI version shown in Figure 23(b) applies the two adaptive cache management techniques in the reverse manner. The Parallel version of IACM simultaneously performs ADI and AWL. As quantified in Section 4.3.2, the ADI-AWL version outperforms the other two versions mainly because it immediately applies ADI, which is highly effective for mitigating instances of performance degradation due to high cache contention and eliminates the potential performance interference between AWL and ADI. Based on this observation, we focus on the design and implementation of the ADI-AWL version throughout the rest of this section.

4.2.2 Adaptive Cache Indexing

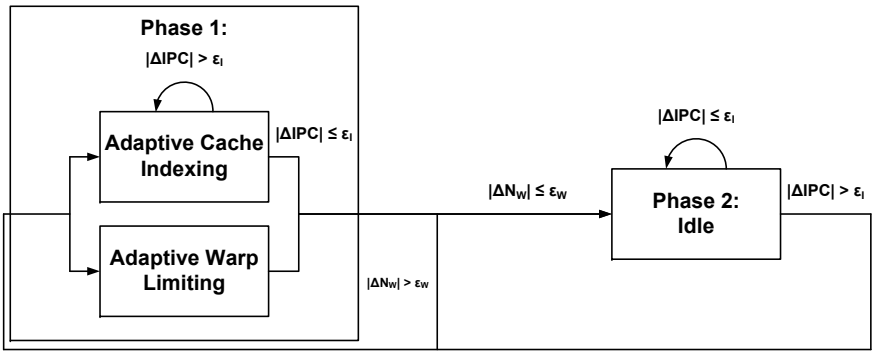
The ADI phase of IACM builds on our prior work [12] that extends ASCIB [32,33], an adaptive indexing technique for CPU caches, in the context of GPGPU computing. We use ASCIB as



(a) The ADI-AWL version



(b) The AWL-ADI version



(c) The Parallel version

Figure 23: The overall execution flow of IACM

the baseline for ADI owing to its simplicity and applicability to L1 caches [32, 33]. Although Section 3.4.3 indicates that ADI has considerable performance gains when applied to L2 caches, we only apply ADI to L1 caches because the performance gains when ADI is applied to L1 caches is more significant compared those when applied to L2 caches, and we want to minimize the hardware overhead. This section focuses on the changes made to the design of ADI for IACM and refer readers to Section 3.3 for more details of ADI.

Figure 24 shows the overall execution flow of ADI. The execution flow has been modified from Figure 6 to perform adaptation only at the ADI phase of IACM and to change phase of IACM when the proper condition is met (i.e., Figure 23). First, when a memory request is sent to the cache, IACM checks the current phase. If the current phase of IACM is not ADI, adaptation will not be performed based on the information of the cache access.

Second, at the end of *idle* sub-phase, IACM checks if the difference in the instructions per cycle (IPC) between the current and previous idle sub-phase is within a threshold (i.e., 25%). If the IPC difference is within the threshold, IACM determines that there is a diminishing gain with ADI and proceeds with the AWL phase. Otherwise, it triggers the victimization sub-phase

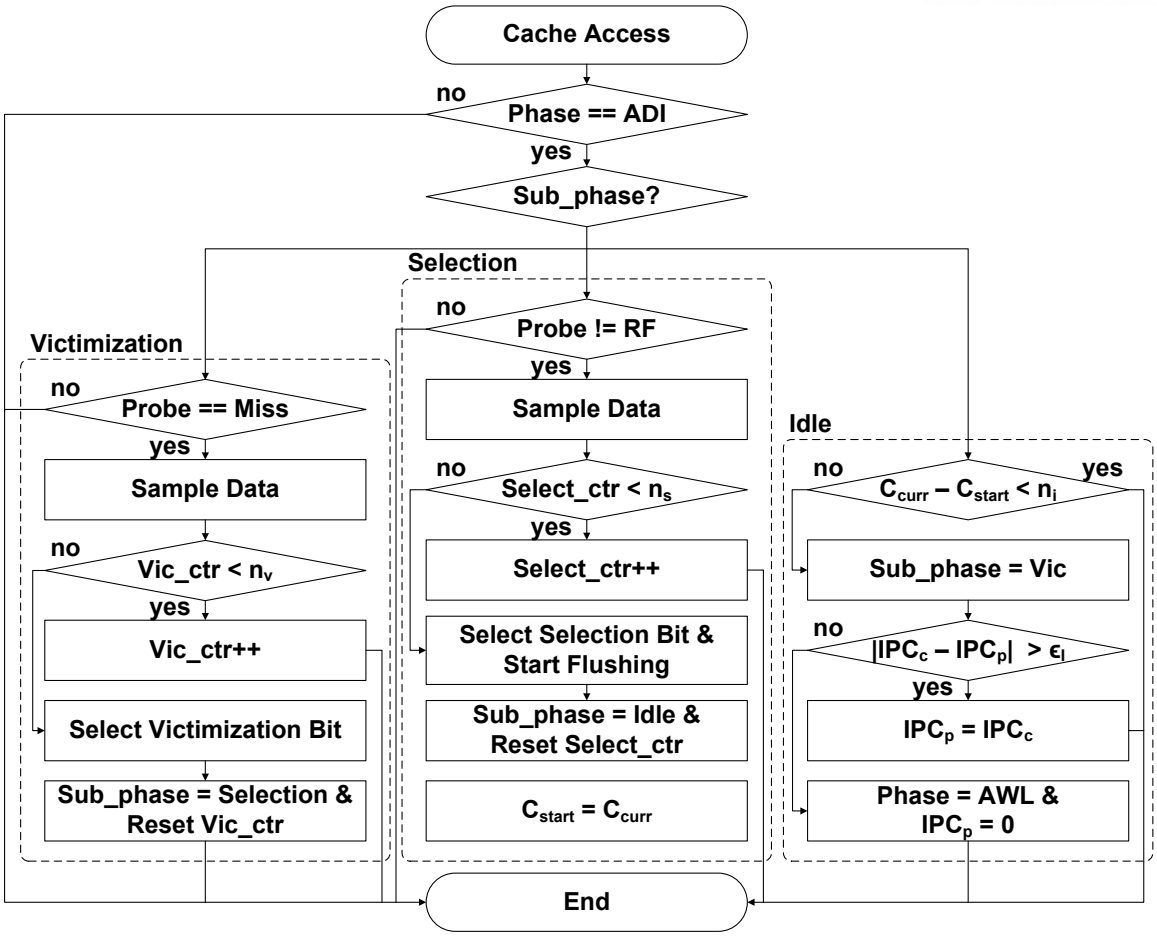


Figure 24: The overall execution flow of ADI

of ADI again to discover indexing bits with higher quality.

Finally, we tweaked design parameters of ADI to achieve optimal performance and energy efficiency when ADI is integrated into IACM. Based on the IACM design space exploration (i.e., Section 4.3.2), the periods for the victimization, selection, and idle sub-phases are set to 250 misses, 250 accesses, and 6250 cycles respectively.

Due to the change in the design parameters of ADI, the hardware overhead also changed. For instance, using the equation derived in Section 3.3, we calculated that for the victimization sub-phase, it requires 120 bits for the counters, 15 8-bit adders, and 10 XOR gates because $S = 5$ and $M = 8$. For the selection sub-phase, 529 bits for the tag cache and counters, 21 8-bit adders, and 21 XOR gates are needed because $N = 32$, $S = 5$, $B = 7$, and $M = 8$. As quantified in Section 4.2.6, the area overhead for ADI is also low.

4.2.3 Adaptive Warp Limiting

After the ADI phase, IACM enters the adaptive warp limiting (AWL) phase. During the AWL phase, IACM periodically samples the IPC of the GPGPU workload and dynamically adapts the active warp count based on the trend of IPC changes to find the optimal warp count that

maximizes the performance. Some earlier works use performance metrics that can be collected using rather complicated hardware structures (e.g., locality scores [3]). Instead, IACM uses IPC as the main performance metric because it can be collected without any extra hardware using the performance counters already available in commercial GPGPU architectures [1]. It also provides complete performance data.

Algorithms 1, 2, 3, and 4 show the pseudocode for AWL. The AWL phase consists of sub-phases – binary search and linear search sub-phases. During the binary search sub-phase (Algorithm 2), IACM searches for the optimal warp count using the binary search algorithm. IACM starts from the current active warp count. First, it halves the warp count and checks if reducing the warp count increases the IPC. If this is the case, IACM continues to halve the warp count after each period (i.e., 6250 cycles) until the IPC stops increasing.[8] Otherwise, it attempts to double the warp count and repeats a similar process. If the binary search reaches the minimum or maximum warp count, the AWL phase terminates. Otherwise, the AWL phase enters the linear search sub-phase.

During the linear search sub-phase (Algorithm 3), IACM searches for the optimal warp count in a fine-grained manner. IACM periodically (i.e., 6250 cycles) samples IPC and decrements (or increments) the active warp count by 1 based on the IPC changes. When an optimal warp count that maximizes IPC is found, the AWL phase terminates.

At the end of the AWL phase, IACM decides whether to proceed with the ADI phase or the idle phase based on the change in the optimal warp count as determined during the AWL phase. If the difference in the previous and current optimal warp counts exceeds a threshold (i.e., 25%), IACM determines that the system has not yet reached a stable state and starts from the ADI phase again (Line 5 in Algorithm 4). Otherwise, IACM enters the idle phase (Line 3 in Algorithm 4).

For the hardware overhead of AWL, several extra registers are needed to track the data (i.e., previous (5 bits) and current (5 bits) warp counts, search mode (i.e., binary or linear (1 bit)), direction (i.e., up or down (1 bit)), etc.). The per-core storage overhead required for the registers is 74 bits.

In addition, some extra hardware logic (e.g., shifters and comparators) is required for AWL. Specifically, 13 multiplexors, 18 comparators, one OR gate, two AND gates, five 5-bit adders, and four 5-bit shifters are required. In line with prior work [3, 4], the logic overhead required for AWL is expected to be insignificant compared to the other logic that already exist in each SIMT core. In addition, as quantified in Section 4.2.6, the area overhead for AWL is also low.

Algorithm 1 Pseudocode for adaptive warp limiting

```
1: procedure ADAPTIVEWARPLIMITING
2:     if phase = AWL then
3:         if N_T = 1 then                                  ▷ N_T: total number of warps
4:             N_C ← 1                                      ▷ N_C: number of active warps
5:             changePhase(N_C)
6:         else if kernelInit = true ∨ awlPhaseInit = true then
7:             if kernelInit = true then
8:                 N_C ← N_T
9:             end if
10:            if N_C = 1 then
11:                direction ← up
12:            else
13:                direction ← down
14:            end if
15:            search ← binary
16:            subPhaseCycles ← 0
17:            IPC_prev ← 0
18:            kernelInit ← false
19:            awlPhaseInit ← false
20:            N_S ← N_C                                    ▷ N_S: records N_C when search begins
21:        end if
22:        if subPhaseCycles > C_P then
23:            subPhaseCycles ← 0
24:            if search = binary then
25:                doBinarySearch()
26:            else                                         ▷ search = linear
27:                doLinearSearch()
28:            end if
29:            subPhaseCycles++
30:        end if
31:    end if
32: end procedure
```

Algorithm 2 Pseudocode for the doBinarySearch function

```
1: procedure doBinarySearch
2:     if IPC_prev < IPC_curr then
3:         IPC_prev ← IPC_curr
4:         if N_C = 1 ∨ N_C ≥ N_T then
5:             changePhase(N_C)
6:         else
7:             N_C ← getNextWarpCount(search, direction, N_C)
8:         end if
9:     else                                          ▷ IPC_prev ≥ IPC_curr
10:        IPC_prev ← 0
11:        if N_C = N_S/2 then
12:            direction ← up
13:            N_C ← N_S
14:        else
15:            if direction = down then
16:                N_C ← 2N_C
17:            end if
18:            N_S ← N_C
19:            direction ← down
20:            search ← linear
21:        end if
22:    end if
23: end procedure
```

4.2.4 Idle Phase

During the idle phase, IACM periodically (i.e., 6250 cycles) samples the IPC of the GPGPU workload without performing any ADI or AWL-related activities. If the difference in the previous and current IPCs exceeds a threshold (i.e., 25%), IACM terminates the idle phase and proceeds with the ADI phase. The per-core storage overhead of the idle phase is 38 bits, which are required for the registers that holds the previous and current IPCs.

[8]The `getNextWarpCount` function invoked at Line 7 in Algorithm 2 and at Line 5 in Algorithm 3 takes three parameters (i.e., the search sub-phase (i.e., binary or linear), the search direction (i.e., up or down), and the current warp count) and computes the next warp count accordingly. For brevity, we omit the code of the `getNextWarpCount` function.

Algorithm 3 Pseudocode for the doLinearSearch function

```
1: procedure doLinearSearch
2:     if IPC_prev < IPC_curr then
3:         IPC_prev ← IPC_curr
4:         if N_C ≠ N_S/2 ∧ N_C ≠ 2 · N_S ∧ N_C ≤ N_T then
5:             N_C ← getNextWarpCount(search, direction, N_C)
6:         else
7:             changePhase(N_C)
8:         end if
9:     else                                          ▷ IPC_prev ≥ IPC_curr
10:        IPC_prev ← 0
11:        if N_C = N_S − 1 then
12:            N_C ← N_S
13:            direction ← up
14:        else
15:            if direction = down then
16:                N_C ← N_C + 1
17:            else
18:                N_C ← N_C − 1
19:            end if
20:            changePhase(N_C)
21:        end if
22:    end if
23: end procedure
```

4.2.5 Cache Bypassing

IACM employs the cache bypassing technique [42] for the L1 data cache. For seamless integration with other adaptive techniques, IACM enables cache bypassing in all three phases. Similar to an earlier technique [4], IACM builds upon the protection distance (PD) information of each cache line to dynamically determine whether a memory request should be bypassed or not. Each cache line is augmented with a few extra bits to track the remaining protection distance (RPD) information dynamically. When a memory object is newly loaded into a cache line, the RPD counter of the cache line is initialized to an initial value (i.e., 4). When the other cache line in the same cache set is accessed, the RPD counter is decremented. When the cache line is accessed again, its RPD counter value is set to the default value.

In IACM, memory requests bypass the L1 data cache if every cache line in the associated set is currently holding a non-zero RPD counter value or reserved for other pending memory requests (i.e., reservation fails). The main purposes of the former and the latter are to protect the existing

Algorithm 4 Pseudocode for the changePhase function

```
1: procedure CHANGEPHASE(N_C)
2:     if |N_C − N_P| ≤ ε_W then
3:         phase ← Idle
4:     else
5:         phase ← ACI
6:     end if
7:     awlPhaseInit ← true
8:     N_P ← N_C
9: end procedure
```

Table 3: Hardware overheads of the IACM components for the baseline GPGPU architecture with 16 SIMT cores

Components	**ADI**	**AWL**	**Idle**	**Cache bypassing**
Storage (bits)	10384	1184	608	6144
Area(mm^2)	1.40×10^{-2}	3.16×10^{-3}	1.74×10^{-3}	7.69×10^{-3}
Area (% to L1D)	1.875	0.459	0.286	1.005
Area (% to GPGPU die)	0.00265	0.000597	0.000329	0.00145

cache lines for reuse and to fully utilize the high interconnection network (ICN) bandwidth of the GPGPU architecture, respectively. IACM prevents the ICN from being oversaturated due to excessive cache bypassing by judiciously controlling the active warp count during the AWL phase.

For the hardware overhead associated with cache bypassing, each cache line is augmented with an N bit counter to track the RPD information. For instance, if the GPGPU architecture uses the four-way set-associative 16KB L1 data cache with 128B cache lines and a 3 bit RPD counter for each cache line, the extra bits needed (per core) to store the RPD information amount to 384 bits (i.e., 128 cache lines, 3 bits per cache line), which is insignificant and in line with the findings of previous work [4].

4.2.6 Hardware Overheads

Table 3 summarizes the hardware overheads of the IACM components. Specifically, we report the storage overheads (in bits) for all 16 SIMT cores to implement each of the IACM components by extending the baseline GPGPU architecture discussed in Section 4.3.1. In addition, we report their area overheads (for all 16 SIMT cores), as estimated using CACTI [43] in terms of both the absolute area (i.e., mm^2) and the percentage of the total area of the L1 data cache and baseline GPGPU architecture [44]. We observe that IACM incurs low hardware overhead.

Algorithm 5 Psuedocode for idlePhase

```
1: procedure IDLEPHASE
2:     if phase = Idle then
3:         if subPhaseCycles > C_P then
4:             subPhaseCycles ← 0
5:             if idlePhaseInit = true then
6:                 idlePhaseInit ← false
7:             else
8:                 if |IPC_curr − IPC_prev| > ε_I then
9:                     phase ← ACI
10:                    idlePhaseInit ← true
11:                end if
12:            end if
13:            IPC_prev ← IPC_curr
14:        end if
15:        subPhaseCycles++
16:    end if
17: end procedure
```

4.3 Evaluation

This section provides a quantitative evaluation of IACM. Our quantitative evaluation has following goals – (1) to assess the performance and energy efficiency of IACM using various GPGPU workloads, (2) to make a comparison with the state-of-the-art technique [6], and (3) to assess the sensitivity of the performance and energy-efficiency gains of IACM to architectural parameters such as the L1 data cache capacity and the interconnection network (ICN) bandwidth.

For cache indexing, we also quantify the effectiveness of the baseline (i.e., conventional 4-way set associative) and advanced static cache indexing (i.e., a variant of bit-wise XOR [10,29]) schemes along with the adaptive cache indexing scheme. Specifically, for the 4-way 16KB L1 data cache with 128-byte blocks, the advanced static indexing scheme computes the indexing bits as follows – $I_4 = A_{19} \oplus A_{11}$, $I_3 = A_{17} \oplus A_{10}$, $I_2 = A_{15} \oplus A_9$, $I_1 = A_{14} \oplus A_8$, and $I_0 = A_{13} \oplus A_7$ [29].

4.3.1 Methodology

We implemented IACM in the GPGPU-Sim simulator (version 3.2.2) [16]. We use the architectural parameters defined in the configuration file in the GTX480 directory (Table 6). To quantify the energy efficiency of IACM, we use GPUWattch [34].

Table 7 summarizes all of the evaluated benchmarks, which are selected from the benchmark suites proposed in earlier studies [28,35–38]. Inspired by an earlier classification proposed in [6], we classify the benchmarks into the five categories based on their architectural characteristics –

Table 4: Simulation parameters

Parameter	Value
SIMT core	Core count: 16, SIMT width: 32, pipeline depth: 5, frequency: 700MHz
Per-core resource	Num. of registers: 32768, scratchpad: 48KB, MSHRs: 32, warps: 48, threads: 1536
Schedulers	Warp scheduler: Greedy-Then-Oldest (GTO), CTA scheduler: round-robin
L1 data cache	Capacity: 16KB/core, line size: 128B, associativity: 4, coalescing: enabled
ICN	Frequency: 700MHz, channel width: 32
L2 cache	Capacity: 768KB, number of banks: 12, line size: 128B, associativity: 8
DRAM	Frequency: 924MHz, scheduler: FR-FCFS, num. of MCs: 6, channel BW: 4B/cycle

the streaming, conflicting, thrashing, conflicting and thrashing, and cache friendly benchmark categories.

In addition, we classify the benchmarks into the three categories based on their average kernel execution cycles (i.e., length), denoted as short, medium, and long. Specifically, each benchmark is sorted into the short, medium, or long benchmark category if its average number of kernel execution cycles with the baseline GPGPU architecture is equal to or shorter than 8×10^5 cycles, longer than 8×10^5 cycles and equal to or shorter than 2×10^7 cycles, or longer than 2×10^7 cycles, respectively.

4.3.2 IACM Design Space Exploration

Prior to quantifying the performance and energy efficiency of IACM, we compare the performances of the three IACM designs (i.e., the ADI-AWL, AWL-ADI, and Parallel versions) discussed in Section 4.2.1 and determine the best IACM design among the three in terms of performance. Because the performance of each IACM design is dependent of the design parameters, we investigate the performance sensitivity and configure it with the design parameters that yield the highest performance. The key design parameters of interest are the ADI period, the AWL period, the threshold for the IPC difference in the ADI phase, and the threshold of the IPC difference in the idle phase.

Figures 25(a), 25(b), 25(c), and 25(d) show the performance sensitivity of each IACM design to the four design parameters. We use the IPC of each IACM design normalized to the baseline design (i.e., higher is better) as a performance metric.

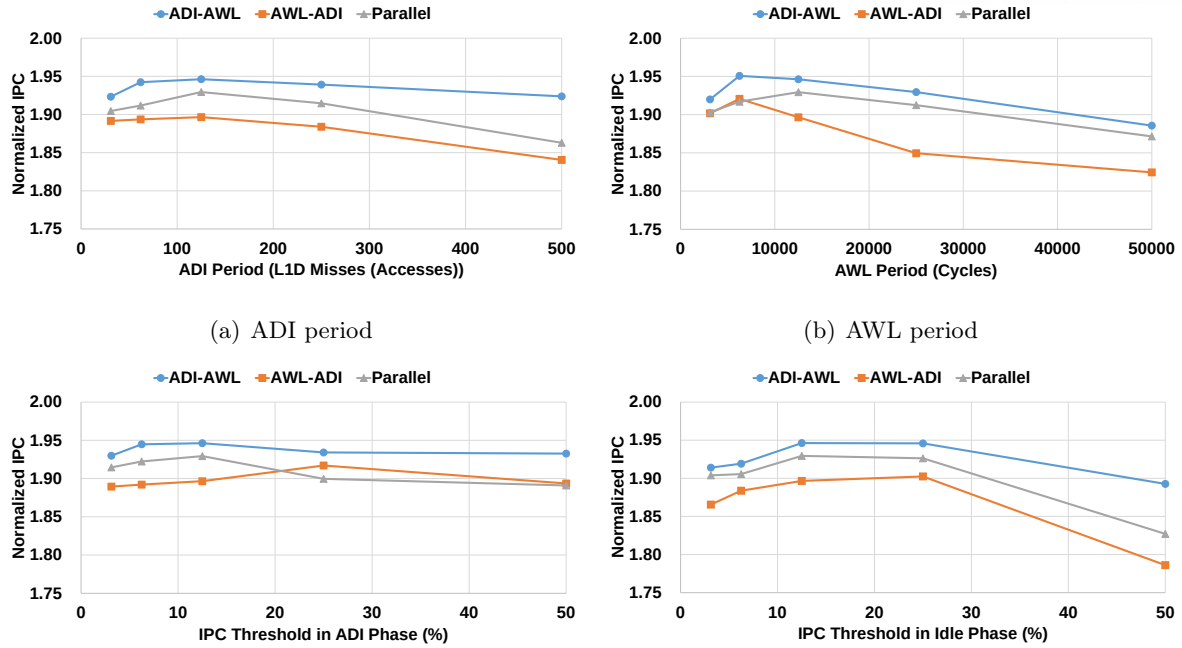
First, in most cases, the performance of each IACM design increases, reaches a maximum,

Table 5: Benchmarks

Category	Name	Description	Length
Streaming	HS	HotSpot [35]	Short
	NW	Needleman-Wunsch [35]	Short
	BLK	Black Scholes [36]	Short
	CONV	Convolution [36]	Medium
	FWT	Fast Walsh Transform [36]	Short
Conflicting	2DC	2D Convolution [28]	Medium
	2MM	2 Matrix Multiplications [28]	Medium
	SRAD	Speckle Reducing Anisotropic Diffusion [35]	Medium
	SC	Streamcluster [35]	Medium
Thrashing	BFS	Breadth-First Search [35]	Short
	KM	Kmeans [35]	Medium
	II	Inverted Index [37]	Medium
	SPMV	Sparse Matrix-Vector Multiplication [38]	Short
C+T	ATAX	Matrix Transpose and Vector Multiplication [28]	Long
	GSM	Scalar, Vector and Matrix Multiplication [28]	Long
	SYRK	Symmetric Rank-K Operations [28]	Long
Friendly	BP	Back Propagation [35]	Short
	BT	B+ Tree [35]	Medium
	NN	Nearest Neighbor [35]	Short
	OP	Monte Carlo Option Pricing [36]	Short

and then decreases as the value of each design parameter increases. This is mainly because there is a tradeoff between the increases and decreases of each of the design parameter values. With small parameter values, IACM performs adaptations with insufficient runtime information (i.e., the ADI and AWL periods), or remains in the ADI phase with diminishing gains (i.e., the IPC threshold used in the ADI phase), or reduces the stability of the system by triggering a re-adaptation too frequently (i.e., the IPC threshold used in the idle phase), resulting in suboptimal performance. With large parameter values, IACM performs adaptation too slowly (i.e., the ADI and AWL periods) or converges to a suboptimal system state (i.e., the IPC thresholds), degrading the overall performance.

Second, the ADI-AWL version outperforms the other two versions in most cases. This occurs because ADI is generally more effective than AWL for mitigating cache contention and improving the overall performance for a subset of the evaluated benchmarks. Therefore, by performing ADI first, the ADI-AWL version has more opportunities for dynamically discovering high-quality indexing bits and quickly adapting to a near-optimal system state than the AWL-ADI version, resulting in higher performance.



(a) ADI period

(b) AWL period

(c) IPC threshold in the ADI phase

(d) IPC threshold in the idle phase

Figure 25: IACM design space exploration

The Parallel version suffers from the interference between ADI and AWL in that the performance effect made by a technique impacts the decision of the other technique in a negative manner. For instance, in a certain adaptation period, AWL could choose a warp count that decreases the overall performance and ADI could select indexing bits that increase the overall performance. If the overall performance impact of ADI on the target application is more significant than that of AWL in that period, the overall performance would be improved. With the improved performance, AWL would continue to change the warp count in the same direction in the subsequent periods, which is highly inefficient. Due to the performance interference between ADI and AWL, the Parallel version of IACM tends to converge to a suboptimal state or take more time to converge to an efficient state. In contrast, ADI-AWL applies the two techniques in a controlled manner, achieving higher performance.

To determine the best IACM design from among the three in terms of performance, we choose the three values of each design parameter for each IACM design that outperform the other values. For instance, we choose to set the AWL period to 6250, 12500, or 25000 cycles for the ADI-AWL version because these values outperform the other values. We then configure each IACM design with 81 different settings (i.e., three values for each of the four design parameters) and determine the best setting that leads to the highest performance for each IACM design.

Because the ADI-AWL version configured with the ADI period of 250 L1 data cache misses (accesses), the AWL period of 6250 cycles, and the IPC thresholds of 25% outperforms the other two versions, we investigate its performance through the rest of the dissertation. With this configuration, the ADI-AWL version spends 6.22×10^4 and 3.23×10^4 cycles on average

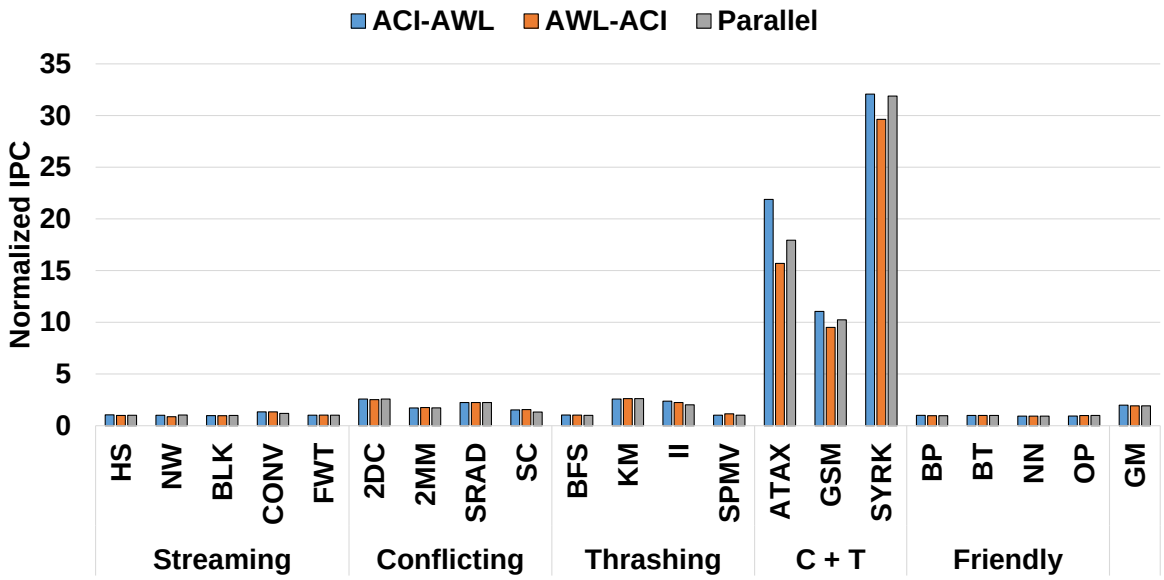
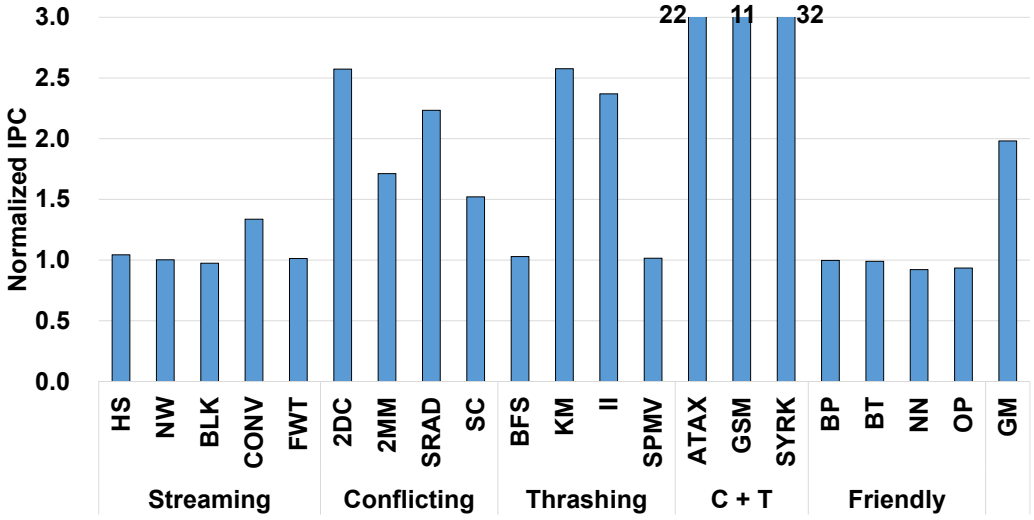


Figure 26: Performance comparison of the three IACM designs

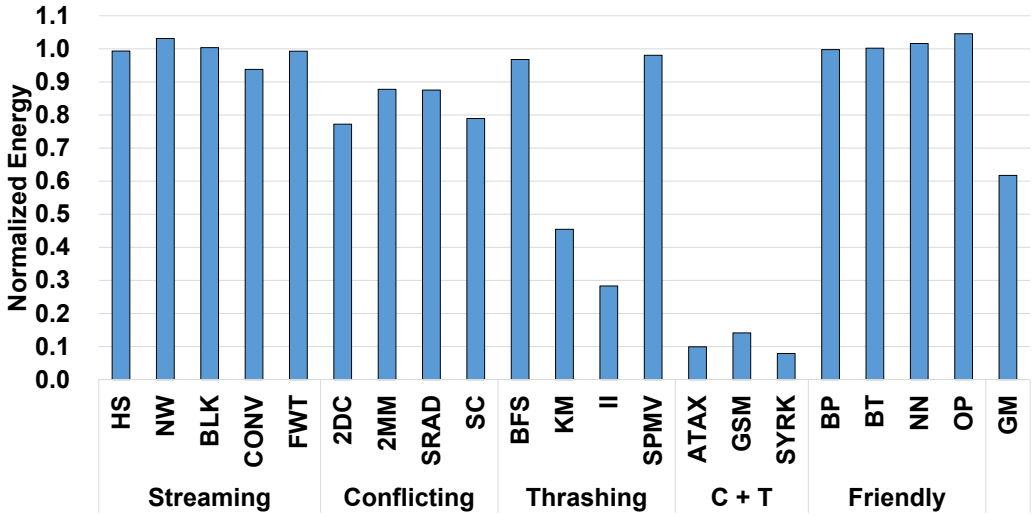
to execute the ADI and AWL phases, respectively. In other words, the time overheads of the ADI and AWL phases are low at 0.42% and 0.22% of the average kernel cycles of the evaluated benchmarks.

To investigate the performance differences of the three IACM designs in more detail, Figure 26 shows the performances of all evaluated benchmarks with each IACM design. Among the evaluated benchmarks, we focus on the performances of the three IACM designs with the conflicting and thrashing (C+T) benchmarks, which exhibit high performance sensitivity to the effectiveness of the cache management techniques used.

We observe that the ADI-AWL version of IACM significantly outperforms the AWL-ADI (i.e., 20.6% on average across the C+T benchmarks) and Parallel (i.e., 9.8% on average across the C+T benchmarks) versions with the C+T benchmarks. The ADI-AWL version achieves significantly higher performance than the AWL-ADI version because the ADI-AWL version addresses the intra- and inter-warp interferences in a more efficient manner. Specifically, the performance issue due to inter-warp interference for the C+T benchmarks tends to become significant only after the performance issue due to intra-warp interference is effectively addressed. The ADI-AWL version initially addresses the intra-warp interference through ADI and then the inter-warp interference through AWL, which allows for IACM to perform adaptations more rapidly and efficiently . In contrast, the AWL-ADI version attempts first to address the inter-warp interference through AWL and then the intra-warp interference through ADI, which leads to inefficient adaptations. The Parallel version is considerably outperformed by the ADI-AWL version due to the interference between ADI and AWL.



(a) Performance



(b) Energy

Figure 27: Overall performance and energy results

4.3.3 Performance and Energy Efficiency

First, we investigate the performance (i.e., instructions per cycle (IPC)) and energy efficiency of IACM. Figure 27 shows the performance and energy normalized to the baseline GPGPU architecture with none of the adaptive cache management techniques applied. IACM significantly improves the performance of the GPGPU workloads with the maximum and average (i.e., geometric mean) speedups of 32 and 2.0, respectively over the baseline architecture. IACM results in higher speedups for the benchmarks in the conflicting, thrashing, and conflicting and thrashing (C+T) categories. With IACM, however, some benchmarks, in this case , `OP`, exhibit slight performance degradation, which will be discussed later.

Figure 27(b) shows that IACM can also effectively reduce the energy consumption of GPGPU

workloads. The corresponding maximum and average energy reduction of IACM compared with the baseline architecture are 92% and 38%. Similar to the performance results, there are a few benchmarks that exhibit slightly increased levels of energy consumption (e.g., `OP` by 4.5%).

As IACM requires additional hardware overhead, we have also evaluate the impact of storage overhead by comparing the performance of baseline GPGPU architecture augmented with IACM and GPGPU with increased L1 data cache size equal to the capacity mentioned in Table 3. Our evaluation shows that IACM significantly outperforms GPGPU with increased L1 data cache size. Therefore, IACM is a better solution compared to increasing L1 data cache size equal to the storage overhead of IACM.

To gain deeper insight on the performance and energy results, we provide detailed cycle and energy breakdowns for each benchmark category. To keep this evaluation concise, we select three benchmarks in each category by omitting some of the benchmarks that show the data trends similar to any of the three selected benchmarks.

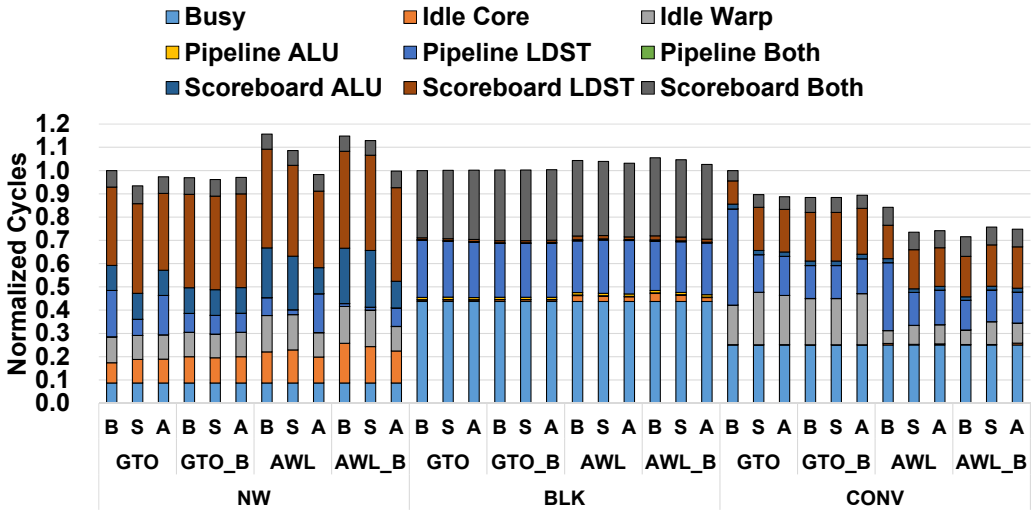
4.3.3.1 Streaming Benchmarks

Figure 28 shows the performance and energy breakdowns of the streaming benchmarks. For each benchmark, we run it with 12 different architectural configurations by changing the cache indexing schemes (i.e., baseline (B), advanced static (S), and ADI (A)), disabling or enabling (*_B) cache bypassing, and disabling (GTO) or enabling adaptive warp limiting (AWL) to investigate the effectiveness of each cache management technique. Note that for each benchmark, the rightmost bar indicates the performance and energy results of IACM.

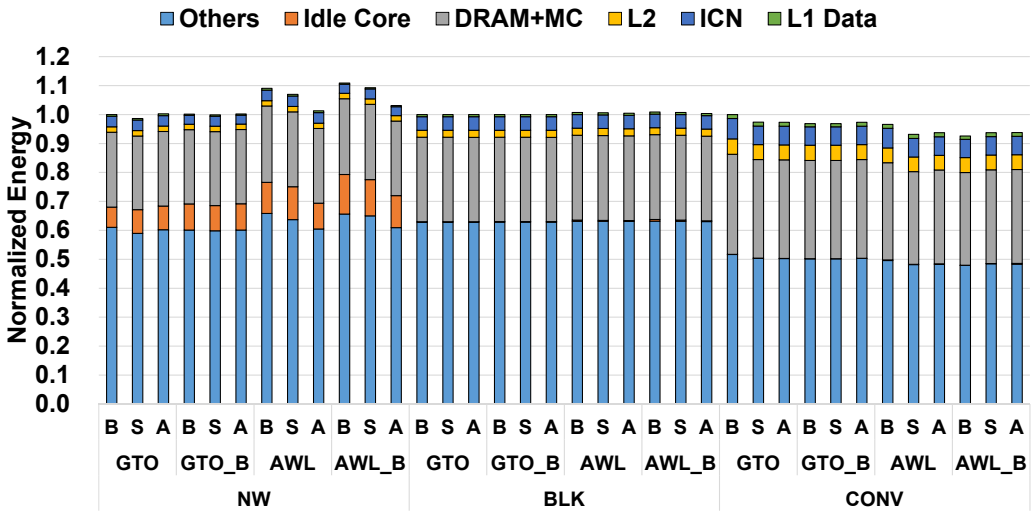
Figure 28(a) shows the cycle breakdown of the streaming benchmarks. Each bar is normalized to the baseline version, in which none of the adaptive cache management techniques is applied. Each bar consists of a number of segments, each of which indicates busy cycles (Busy), idle cycles due to a load imbalance across SIMT cores (Idle Core), idle cycles spent when no warp is ready to execute (Idle Warp), cycles spent for ALU (Pipeline ALU), LDST (Pipeline LDST), and both (Pipeline Both) pipeline stalls, and cycles stalled at the scoreboard while waiting for the data produced by ALU (Scoreboard ALU), LDST (Scoreboard LDST), and both (Scoreboard Both) instructions.

Figure 28(a) demonstrates that IACM improves the performance of `CONV`. This performance gain is mainly due to the decreased latency of the load instructions through a more effective use of the L1 data cache with ADI and AWL.

However, the performance of `BLK` is slightly degraded when AWL is enabled. This is mainly due to the short execution cycles (i.e., 6.71×10^5 cycles on average) of the kernels of `BLK`. The kernels of `BLK` exhibit the best performance with a maximum active warp count. With AWL enabled, IACM temporarily executes `BLK` with suboptimal warp counts (smaller than the maximum warp count) until it discovers the optimal warp count. Since the execution cycles of the kernels of `BLK` are rather short, this overhead is insufficiently amortized, resulting in slight



(a) Performance



(b) Energy

Figure 28: Performance and energy breakdowns of the streaming benchmarks

performance degradation. Because hardware caches provide minor or no performance benefits to `BLK` due to the low locality in memory accesses, ADI and cache bypassing have little performance impact on `BLK`.

Due to reasons similar to those for `BLK` and the short execution cycles (i.e., 2.70×10^4 cycles on average) of the kernels, the performance of `NW` is also degraded when only AWL is enabled. However, ADI mitigates the negative performance effect of AWL on `NW`, resulting in no performance degradation. GTO with advanced static cache indexing performs well with `NW` because the advanced static cache indexing scheme is efficient enough for `NW`.

Figure 28(b) shows the energy consumption outcomes normalized to the baseline version for the streaming benchmarks. The L1 Data Cache, ICN, L2, DRAM+MC, Idle Core, and Others

(mainly related to the SIMT cores) segments indicate the energy consumed by the corresponding hardware components. IACM reduces the energy consumption of `CONV` mainly due to the decrease in the total number execution cycles. IACM slightly increases the energy consumption of `NW` and `BLK` due to the increased execution cycles. However, the increase in the energy consumption of `BLK` is smaller than the increase in the total number execution cycles because most of the additional cycles are idle cycles and some portions of the total energy consumption remain unaffected by IACM. Similarly to the performance trend, the energy efficiency of `NW` is also degraded when only AWL is enabled. However, ADI mitigates the negative effect of AWL, resulting in slight degradation of theenergy-efficiency.

4.3.3.2 Conflicting Benchmarks

Figure 29(a) shows the performance results of the conflicting benchmarks. Compared to the streaming benchmarks, the conflicting benchmarks exhibit higher performance gains with IACM. This occurs because the non-streaming memory access patterns of the conflicting benchmarks can better utilize the L1 data cache when the contention is significantly reduced through the adaptive cache management techniques (especially ADI). We also observe that some of the adaptive cache management techniques are rather ineffective for certain benchmarks. For instance, for `SC`, ADI is effective, whereas AWL is rather ineffective. Nevertheless, IACM significantly improves the performance of `SC` by robustly employing the adaptive cache management techniques.

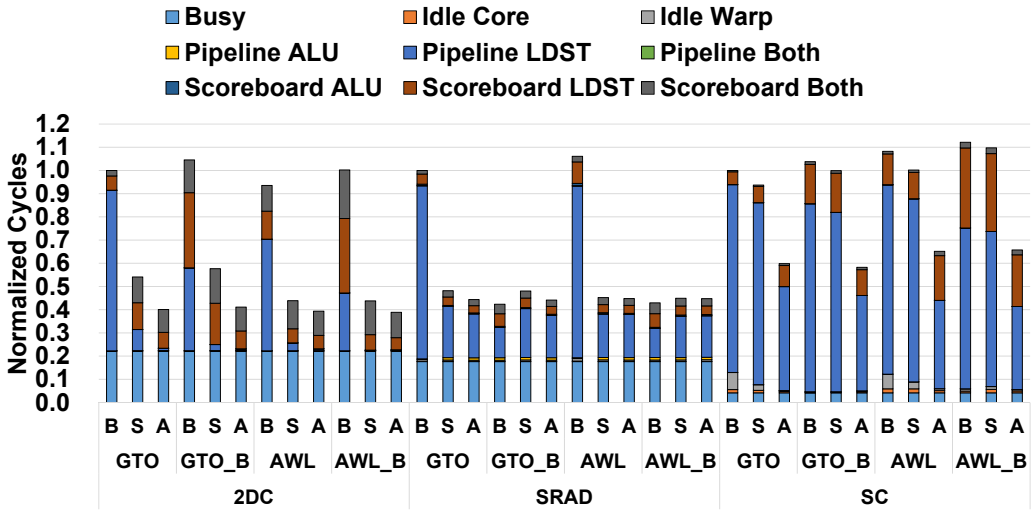
Figure 29(b) demonstrates that IACM significantly reduces the energy consumption of the conflicting benchmarks. In contrast to the case with the streaming benchmarks, IACM reduces the energy consumption in the ICN and L2 cache through better utilization of the L1 data cache through the adaptive cache management techniques.

4.3.3.3 Thrashing Benchmarks

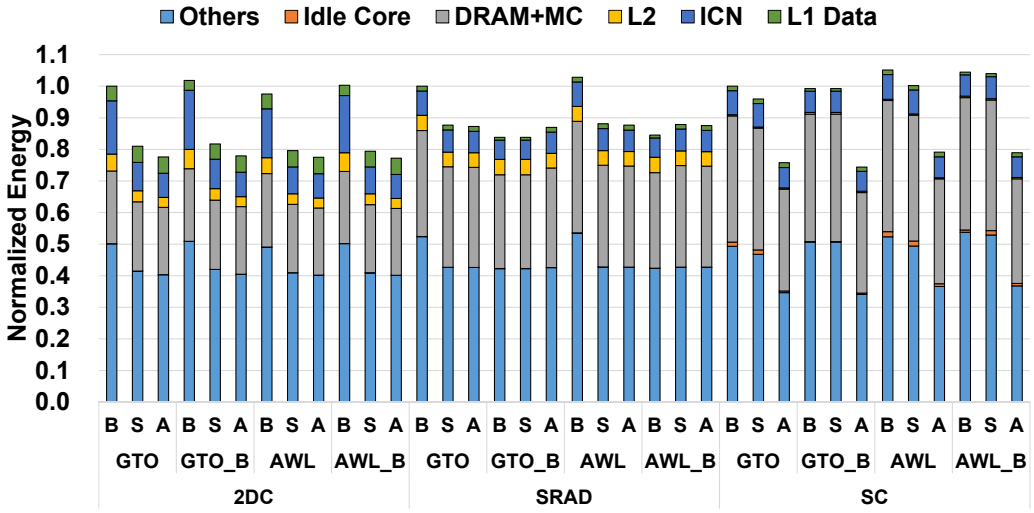
Figure 30(a) shows the performance results of the thrashing benchmarks. AWL is the most effective technique for the thrashing benchmarks because it dynamically adapts the active warp count to avoid cache thrashing. For `II`, ADI and cache bypassing techniques are also effective and constructively composed, allowing IACM to outperform the other architectural configurations.

Figure 30(b) demonstrates that IACM significantly reduces the energy consumption of the thrashing benchmarks, especially `KM` and `II`. IACM reduces the energy consumption of the hardware components (e.g., ICN, L2 cache, memory controllers, and DRAM) in the memory hierarchy by effectively utilizing the L1 data cache through the adaptive cache management techniques.

4.3.3.4 Conflicting and Thrashing Benchmarks



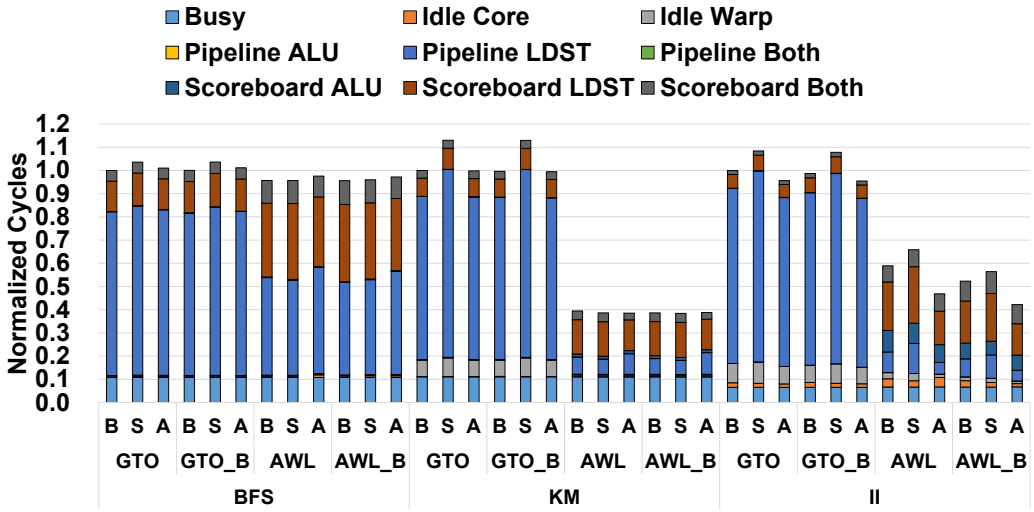
(a) Performance



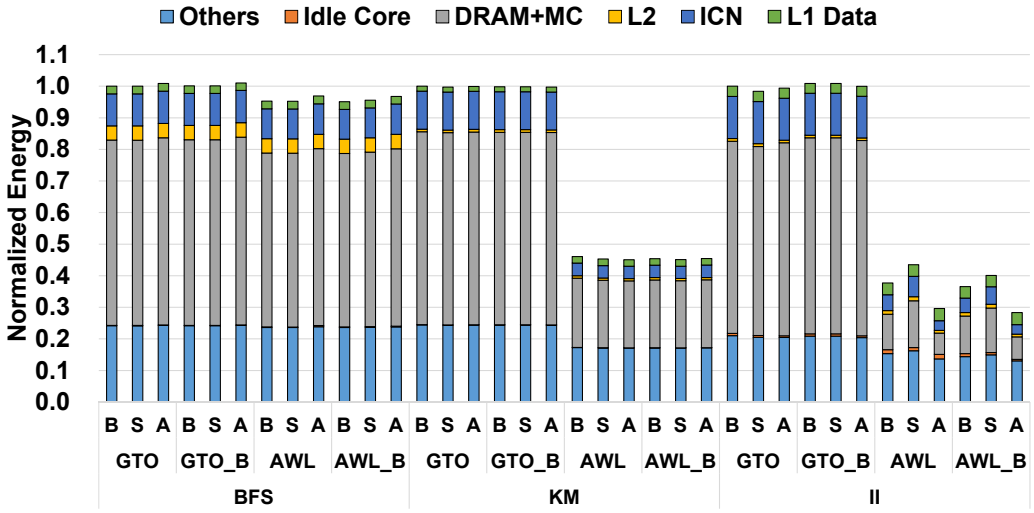
(b) Energy

Figure 29: Performance and energy breakdowns of the conflicting benchmarks

Figure 31(a) shows the performance results of the conflicting and thrashing benchmarks. For `ATAX` and `GSM`, all adaptive cache management techniques effectively improve the performance and are constructively composed, allowing IACM to significantly outperform the other architectural configurations. In particular, ADI significantly outperforms the advanced static cache indexing (the bars labeled as "S") for `ATAX` and `GSM` by dynamically exploiting higher quality indexing bits based on the runtime information, demonstrating the effectiveness of the adaptive approach of IACM. For all conflicting and threshing benchmarks, ADI becomes significantly more effective when AWL is enabled because AWL effectively mitigates thrashing through judicious concurrency control. These performance trends clearly demonstrate the importance of the integrated approach of IACM. Further, Figure 31(b) demonstrates that IACM significantly im-



(a) Performance



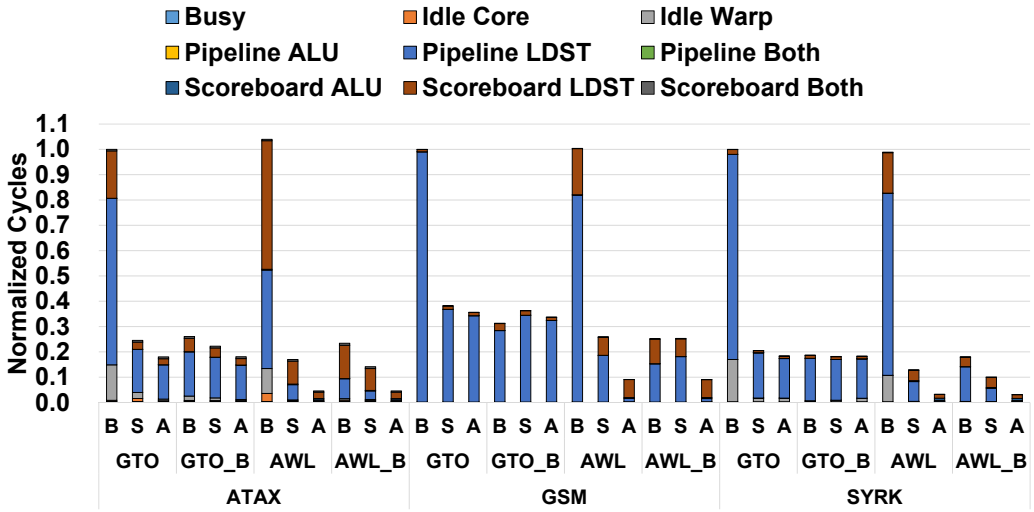
(b) Energy

Figure 30: Performance and energy breakdowns of the thrashing benchmarks

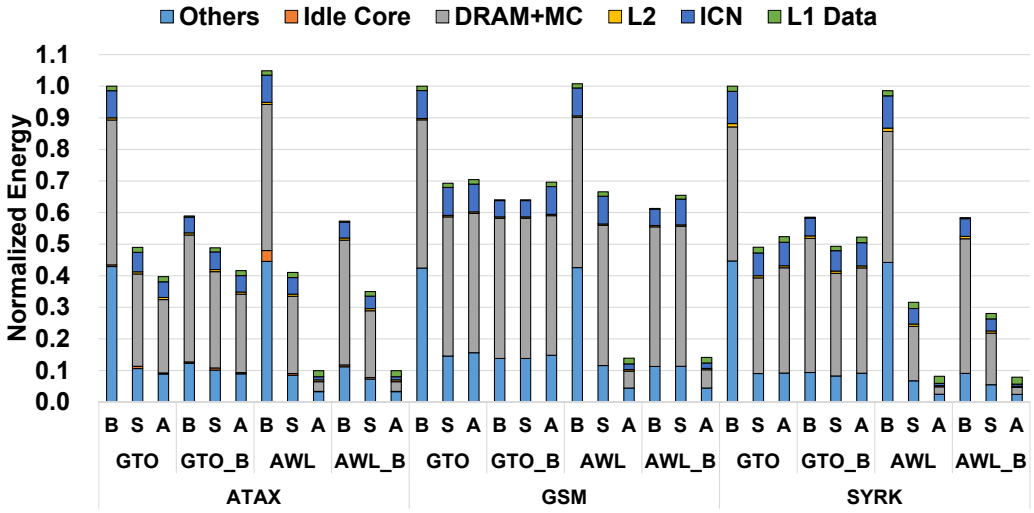
proves the energy efficiency for the conflicting and thrashing benchmarks by effectively reducing the energy consumption of the cores and memory hierarchy components.

4.3.3.5 Cache-Friendly Benchmarks

Figure 32 shows that IACM provides little or no performance and energy-efficiency gain for the cache friendly benchmarks because the baseline version already effectively utilizes the L1 data cache. The performance degradation of `OP` is somewhat higher than those of other benchmarks. Similar to the case with `BLK` in the streaming benchmark category, this occurs mainly because `OP` consists of relatively short kernels (i.e., 7.55×10^5 cycles on average), with which the overhead of IACM may not be fully amortized, resulting in performance degradation



(a) Performance



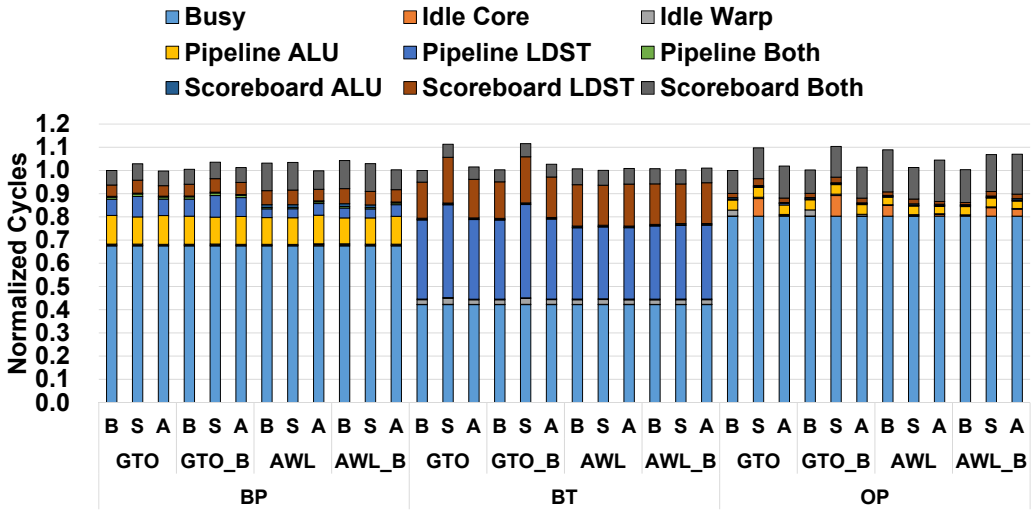
(b) Energy

Figure 31: Performance and energy breakdowns of the conflicting and thrashing benchmarks

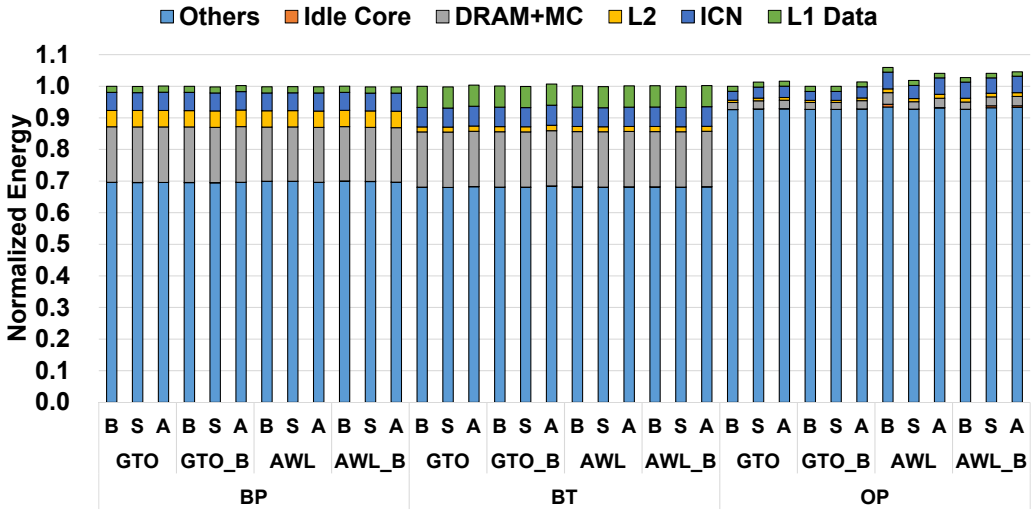
of 7.0%.

4.3.3.6 Summary of the Performance and Energy Results

Figure 33 shows the overall performance and energy results of different combinations of the advanced cache management techniques across all the evaluated GPGPU workloads. We observe the following data trends. First, IACM significantly outperforms the other versions in which only one or two advanced cache management techniques are applied, which demonstrates the effectiveness of the integrated approach that IACM employs. Second, ADI and AWL are rather more effective than cache bypassing for improving the performance and energy efficiency of GPGPU workloads. Finally, ADI provides significant performance and energy-efficiency gains



(a) Performance



(b) Energy

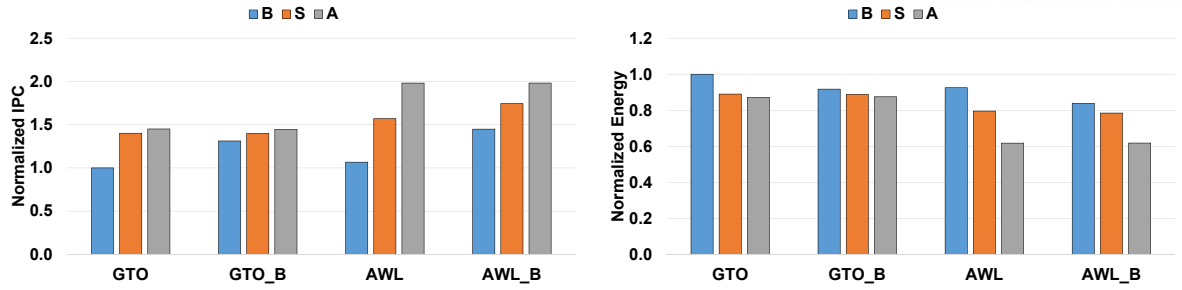
Figure 32: Performance and energy breakdowns of the cache-friendly benchmarks

over the static advanced cache indexing scheme by effectively utilizing high quality indexing bits guided on the runtime information.

4.3.4 Comparison with the State-of-the-Art Technique

We compare IACM with the state-of-the-art technique [6], which applies advanced static cache indexing, warp limiting, and cache bypassing. Figure 34 shows the normalized IPC of IACM and the state-of-the-art technique (i.e., `DWT-PRIC`) with the benchmarks in the streaming, conflicting, thrashing, and C+T categories. We observe the following data trends.

First, IACM significantly outperforms `DWT-PRIC` with various benchmarks (e.g., `CONV`, `KM`, `II`, `GSM`, and `SYRK`). This is mainly because IACM dynamically finds and employs the best indexing



(a) Performance (b) Energy

Figure 33: Performance and energy results of different combinations of the advanced cache management techniques

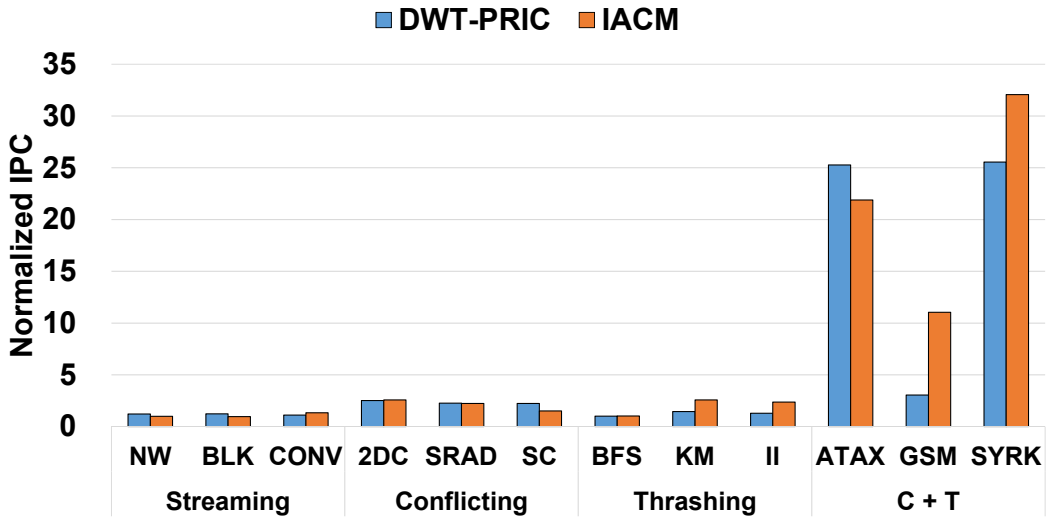


Figure 34: Performance comparison with the state-of-the-art technique

bits for each benchmark based on the runtime information (e.g., `GSM` and `SYRK`), the adaptive warp limiting of IACM works more efficiently than the dynamic warp throttling of DWT-PRIC that suffers from the sampling noise caused by contention on the hardware resources (e.g., L2 cache, ICN) shared among the SIMT cores[9] (e.g., `KM` and `II`) or the use of a decision mechanism based on cache misses per instruction, which occasionally makes the ineffective decision of disabling dynamic warp throttling (e.g., `CONV`), and/or IACM applies the three adaptive techniques in a more coordinated manner.

Second, IACM is rather outperformed by `DWT-PRIC` with some benchmarks (e.g., `BLK`, `SC`,

[9]For concurrency control, DWT-PRIC executes a portion of a kernel by configuring the warp count of each SIMT core differently during the initial sampling phase, determines the SIMT core with the highest performance, and executes the remaining portion of the kernel by setting the warp count of each SIMT core to the warp count (i.e., N_W) of the SIMT core with the highest performance [6]. Due to the contention on the hardware resources (e.g., L2 cache, ICN) shared among the SIMT cores configured with different warp counts during the initial sampling phase, the warp count (i.e., N_W) determined by DWT-PRIC can be inaccurate and inefficient, degrading the overall performance.

and ATAX). This occurs mainly because these benchmarks consist of relatively short kernels (e.g., 6.71×10^5, 2.33×10^6, and 1.72×10^6 cycles for BLK, SC, and ATAX, respectively on average when they are executed with IACM), with which the overhead of IACM may not be fully amortized. IACM requires longer kernel execution cycles to outperform DWT-PRIC compared to the baseline GPGPU architecture because DWT-PRIC is significantly more efficient than the baseline GPGPU architecture. This causes IACM to require longer kernel execution cycles to amortize the cycles spent for its dynamic system state space exploration. However, because the execution cycles of kernels are sufficiently long (especially in non-simulation settings) in common cases, the performance overhead of IACM is insignificant. Third, IACM and DWT-PRIC perform similarly to the benchmarks (e.g., SRAD) for which the static cache indexing of DWT-PRIC is effective.

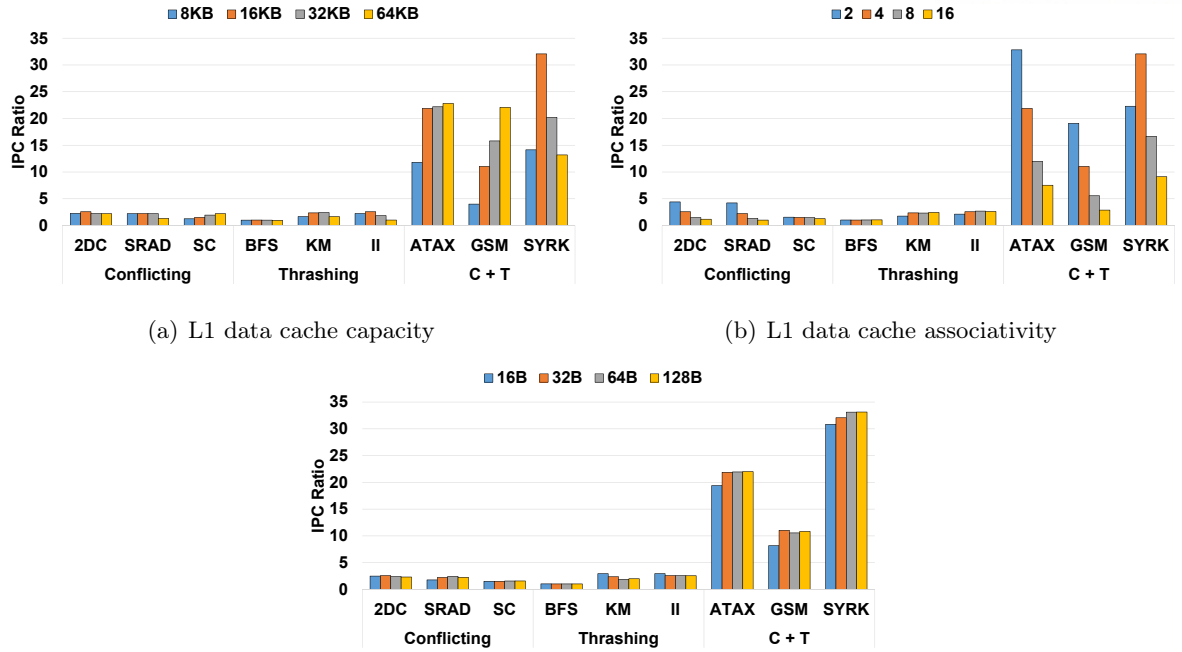
Overall, our experimental results show that IACM achieves considerably higher performance (i.e., 361.4% at a maximum and 7.7% on average) than the state-of-the-art technique (i.e., DWT-PRIC) across all 20 evaluated benchmarks, demonstrating effectiveness of IACM. While detailed results are omitted for conciseness, our quantitative evaluation also shows that the AWL-ADI and Parallel versions of IACM exhibit the performance trends similar to those of the ADI-AWL version for each of the evaluated benchmarks and achieve correspondingly 6.0% and 5.4% higher performance on average than DWT-PRIC across all 20 evaluated benchmarks. The performance gains of the AWL-ADI and Parallel versions are lower than that of the ADI-AWL version of IACM due to their inefficiencies, as discussed in Section 4.3.2.

4.3.5 Sensitivity to Architectural Parameters

With the advancement of architectural and device technologies, the capacity, associativity, and bandwidth of the hardware components in the GPGPU memory hierarchy are expected to continue to increase. Therefore, it is important to investigate the sensitivity of the performance and energy-efficiency gains of IACM to the memory-related architectural parameters. Specifically, we choose to investigate the sensitivity of IACM to the L1 data cache capacity, associativity and the interconnection network (ICN) bandwidth owing to their importance in determining the overall performance and energy efficiency of GPGPU architectures.

Figure 35(a) shows the sensitivity of the performance gain of IACM to the L1 data cache capacity when sweeping it from 8KB to 64KB. The performance gain is defined as the ratio of the IPC of IACM to that of the baseline architecture when the two architectures are configured with the same value of the architectural parameter of interest. We investigate the sensitivity of the benchmarks in the conflicting, thrashing, and conflicting and thrashing categories due to their higher performance sensitivity levels to the architectural parameters compared to other benchmarks categories.

The performance gain of IACM tends to decrease when the L1 data cache capacity exceeds a certain threshold for a subset of benchmarks. For instance, the performance gain of IACM for SYRK continues to decrease when the L1 data cache capacity is 32KB or larger. Because the



(a) L1 data cache capacity

(b) L1 data cache associativity

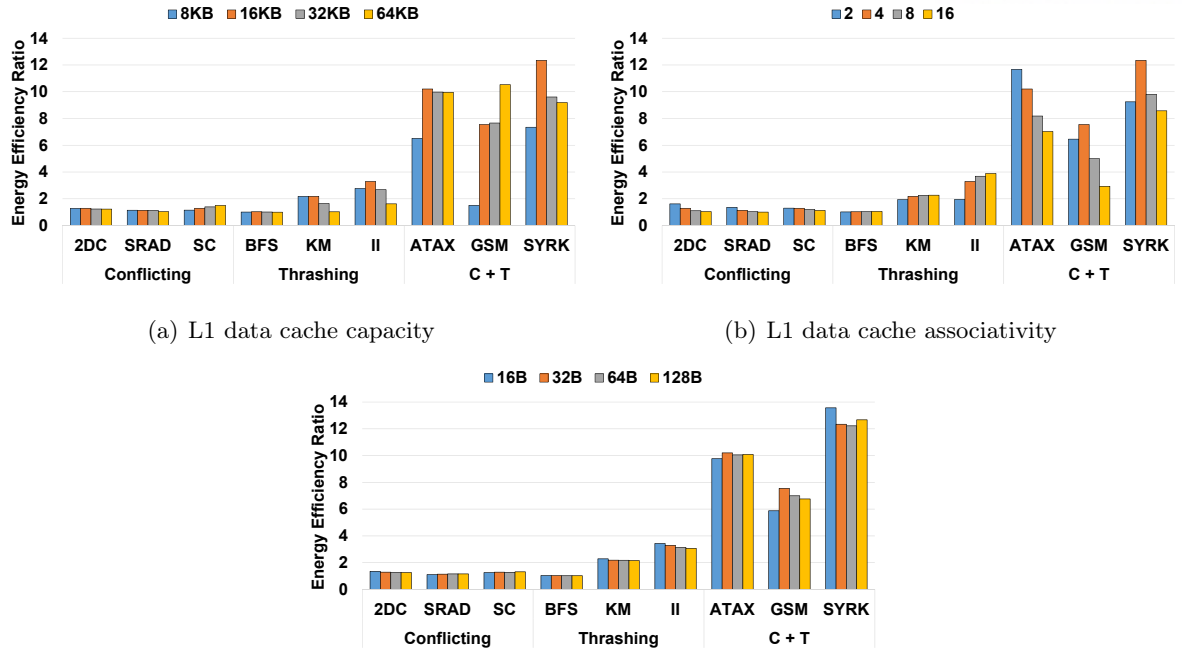
(c) ICN bandwidth

Figure 35: Sensitivity of the performance gain of IACM to architectural parameters

performance pathologies of the baseline are mitigated with a sufficiently large L1 data cache. Interestingly, the performance gain of IACM for cetain benchmarks such as `ATAX`, and `GSM` continues to increase up to the L1 data capacity of 64KB. Because high quality indexing bits for these benchmarks are not located in the low N bits of the cache-block address, increasing the L1 data cache capacity fails to mitigate the performance degradation with the baseline architecture, which uses the low N bits for indexing. In contrast, because IACM robustly resolves the intra-warp interference and effectively utilizes the increasing capacity of the L1 data cache with high quality indexing bits, its performance gain continues to increase.

Figure 35(b) shows the sensitivity of the performance gain of IACM to the associativity of the L1 data cache by sweeping it from 2 to 16. Generally, the performance gain of IACM tends to decrease with higher associativity. In particular, unlike the sensitivity trend with the L1 data cache capacity, the performance gain of IACM for `ATAX` and `GSM` continues to decrease as the associativity increases. Because the intra-warp interference, which drastically degrades the performance of `ATAX` and `GSM` on the baseline architecture can be mitigated with higher associativity. Interestingly, the performance gain of IACM for the thrashing benchmarks increases with higher associativity. Because the thrashing benchmarks incur frequent capacity misses, the baseline architecture does not benefit from high associativity. In contrast, IACM avoids cache thrashing by dynamically controlling the active warp count. Therefore, IACM can fully benefit from high associativity, resulting in gradual increase of performance gains compared to the baseline architecture.

Figure 35(c) shows the sensitivity of the performance gain of IACM to the ICN bandwidth



(a) L1 data cache capacity

(b) L1 data cache associativity

(c) ICN bandwidth

Figure 36: Sensitivity of the energy-efficiency gain of IACM to architectural parameters

when by sweeping the channel width from 16 to 128 bytes. For thrashing benchmarks such as `KM`, the performance gain of IACM decreases with a higher ICN bandwidth, mainly because the baseline architecture suffers from ICN bandwidth saturation due to the thrashing in the L1 data cache. In contrast, IACM does not suffer from ICN bandwidth saturation through the adaptive concurrency control performed by AWL. Therefore, the baseline architecture benefits from the higher ICN bandwidth more significantly than IACM, resulting in a decreasing performance gain.

For the conflicting and conflicting and thrashing benchmarks, the performance gain of IACM increases with a higher ICN bandwidth, mainly because the baseline architecture is bottlenecked by the contention in the L1 data cache, resulting in little or no performance improvement from higher ICN bandwidth.

Figure 36 shows the sensitivity of the energy-efficiency gain (i.e., higher is better) of IACM to the memory-related architectural parameters. IACM exhibits energy-efficiency gain data trends similar to the performance gain data trends and achieves significant energy-efficiency gains across various architectural configurations. This is mainly because IACM achieves significant performance gains across various architectural configurations and performance is one of the major factors that determine the overall energy consumption. We also observe that the energy-efficiency gains of IACM are smaller than the performance gains because some portions (e.g., the energy consumption of the SIMT cores) of the total energy consumption remain unaffected by IACM.

In summary, our quantitative evaluation shows that IACM is promising in the sense that it

can effectively improve the performance and energy efficiency of various GPGPU benchmarks with current technology and continues to deliver significant performance and energy-efficiency gains even when the GPGPU memory hierarchy is enhanced with more advanced technologies.

Figure 37: Overall architecture of BLPP

V BLPP: Improving the Performance of GPGPUs with Heterogeneous Memory through Bandwidth- and Latency-Aware Page Placement

5.1 Motivation

To achieve the best possible performance on GPGPUs with heterogeneous memory, it is crucial to place memory pages across the memory nodes in a heterogeneity-aware manner. The conventional memory placement techniques (e.g., the local and interleave memory placement policies) for non-uniform memory access (NUMA) systems cannot achieve optimal performance because they place memory pages across memory nodes without considering the performance heterogeneity of the memory nodes.

Heterogeneous memory hierarchies exhibit different characteristics in terms of bandwidth and latency. Typically, the GPU memory hierarchy achieves higher bandwidth with GDDR5 technology and lower latency with L1-data and L2 caches. In contrast, the CPU memory hierarchy provides larger capacity with DDR4 technology but lower bandwidths and higher latency levels.

A memory management system for GPGPUs with heterogeneous memory should be able to (1) identify the performance characteristics (e.g., bandwidth sensitive) of the target application accurately and (2) place pages across heterogeneous memory nodes based on the optimal allocation ratio. The state-of-the-art memory-management technique for GPGPUs with heterogeneous memory [17] has limitations in that it (1) neglects the performance effects of GPGPU caches in terms of bandwidth and latency and (2) places memory pages in a latency-oblivious manner. As quantified in this work, the state-of-the-art technique often achieves suboptimal performance owing to such limitations.

5.2 Design and Implementation

BLPP is a type of system software (e.g., OS or runtime system), consisting of following three components (Figure 37) – (1) the application monitor, (2) the allocation ratio controller, and (3) the memory allocator.

5.2.1 Application Monitor

The application monitor monitors the target application to analyze its performance characteristics. During the execution of the target application, the application monitor records the performance counter data (i.e., L1 data ($m_{G,L1D}$) and L2 cache ($m_{G,L2}$), the miss rates, and the data traffic from/to the memory partitions (B_M)). These values are used to identify the performance characteristics of the target application, specifically the memory bandwidth and latency sensitivity.

We present two versions of BLPP – the static (S-BLPP) and dynamic (D-BLPP) versions of BLPP. With S-BLPP, the application monitor collects the performance counter data during the offline profiling process for the target application. The advantage of S-BLPP is that it can place pages across the heterogeneous memory nodes in an optimal manner using the predetermined allocation ratio from the very beginning of the execution process. The main disadvantage of S-BLPP is that it requires offline profiling.

With D-BLPP, the application monitor periodically records the performance counter data at runtime. The memory allocation ratio controller, which is discussed below, computes the optimal memory allocation ratio based on the updated runtime data. The major advantage of D-BLPP is that it requires no offline profiling process. The major disadvantage of D-BLPP is that it may allocate pages across heterogeneous memory nodes with a suboptimal allocation ratio before the optimal allocation ratio is dynamically computed.

5.2.2 Memory Allocation Ratio Controller

Based on the performance data collected from the application monitor, the memory allocation ratio controller determines the optimal allocation ratio across the heterogeneous memory nodes to achieve high performance. BLPP largely classifies target applications into bandwidth-sensitive and other application categories.

BLPP determines the optimal allocation ratio (i.e., $p_{OPT,S}$) for bandwidth-sensitive applications based on the bandwidth of the GPU (B_G) and CPU (B_C) memory nodes [17, 45]. We consider a bandwidth-sensitive application whose total execution time is largely determined by the data (D) transfer time between the SIMT cores and the memory nodes. We assume that the corresponding ratios of the memory allocated to the GPU and CPU memory nodes are p and $1-p$. The total execution time of the target application is then computed using Equation 1, where $t_G = \frac{p \cdot D}{B_G}$ and $t_C = \frac{(1-p) \cdot D}{B_C}$.

$$t_T = \max(t_G, t_C) \tag{1}$$

The total execution time of the bandwidth-sensitive application is minimized if and only if $t_G = t_C$. If $t_G > t_C$, we can continue to reduce the total execution time by decreasing $p >$ until t_G and t_C become equal (and vice versa). With $t_G = t_C$, the optimal ratio ($p_{OPT,S}$) of the data placed in the GPU memory node for bandwidth-sensitive applications is computed using

Equation 2.

$$p_{OPT,S} = \frac{B_G}{B_G + B_C} \tag{2}$$

For bandwidth-insensitive applications, BLPP places memory pages by considering the effective latency of each memory node. Specifically, the effective latency of the GPU memory hierarchy is computed using Equation 3, where $L_{G,L1D}$, $L_{G,L2}$, and $L_{G,M}$ denote the access latencies of the GPGPU L1 data cache, L2 cache, and physical memory, respectively (see Table 6 for the exact latencies of the simulated architecture).

$$L_G = L_{G,L1D} + m_{G,L1D} \cdot (L_{G,L2} + m_{G,L2} \cdot L_{G,M}) \tag{3}$$

The effective latency of the CPU memory node is computed as follows – $L_C = L_{C,M}$, where $L_{C,M}$ denotes the access latency of the CPU physical memory and the physical link between the GPU and CPU. Because the baseline GPGPU architecture disallows the caching of memory objects in the CPU physical memory, the effective latency of the CPU memory node is solely determined by the latency of the CPU physical memory and the GPU-CPU link.

Given that the effective latency of the GPU memory hierarchy is considerably lower than that of the CPU memory node, BLPP sets the optimal allocation ratio (i.e., $p_{OPT,I}$) of the data placed in the GPU memory node for bandwidth-insensitive applications using Equation 4.

$$p_{OPT,I} = 1 \tag{4}$$

In Equation 5, we define the ratio (i.e., r_L) of the average memory access latency with the pages allocated across the heterogeneous memory nodes based on the optimal allocation ratio (i.e., $p_{OPT,S}$) for bandwidth-sensitive applications to the average memory access latency with all pages allocated based on the optimal allocation ratio (i.e., $p_{OPT,I}$) for bandwidth-insensitive applications. If the memory latency ratio (i.e., r_L) of the target application is high, this indicates that the latency of the GPU memory hierarchy is significantly lower than that of the CPU memory node because the target application effectively utilizes the L1 data cache and/or L2 cache in the GPU memory hierarchy.

$$r_L = \frac{\frac{B_G}{B_G+B_C} \cdot L_G + \frac{B_C}{B_G+B_C} \cdot L_C}{L_G} \tag{5}$$

BLPP classifies the target application into the bandwidth-sensitive application category if r_L is lower than a threshold (i.e., θ_L) and the data traffic (i.e., B_M) between the SIMT cores and the memory partitions is higher than a threshold (i.e., θ_B).[10] The rationale behind this decision is that the use of the CPU memory node is beneficial only for applications which fail to effectively utilize the caches in the GPU memory hierarchy and incur a large amount of the

[10]Based on design space exploration with various applications and parameter settings, we set θ_L and θ_B to 1.8 and 140 GB/s, correspondingly. Which achieve the highest performance across all applications and parameter settings.

Algorithm 6 Pseudocode for page placement

```
1:  procedure BLPPAlloc
2:      node ← initialNode
3:      r ← getRandom(0, 1)
4:      if r < p_OPT then
5:          node ← nodeGPU
6:      else
7:          node ← nodeCPU
8:      end if
9:      if node.isFull = true then
10:         otherNode ← getOtherNode(node)
11:         if otherNode.isFull = false then
12:             node ← otherNode
13:         end if
14:     end if
15:     page ← allocPage(node)
16:     return page
17: end procedure
```

data traffic. For instance, if the target application effectively utilizes the L1 data cache with a low miss rate, the use of the CPU memory may significantly degrade the overall performance as memory requests to the CPU memory cannot employ L1 data cache.

Finally, based on the classification result, BLPP determines the optimal allocation ratio (p_{OPT}) for the target application using Equation 6.

$$p_{OPT} = \begin{cases} p_{OPT,S} & \text{if } r_L < \theta_L \wedge B_M > \theta_B \\ p_{OPT,I} & \text{otherwise} \end{cases} \tag{6}$$

5.2.3 Memory Allocator

During the execution of the target application, the memory allocator dynamically allocates pages on demand across the heterogeneous memory nodes based on the optimal allocation ratio determined by the memory allocation ratio controller. Algorithm 6 shows the pseudocode for the memory allocator.

The memory allocator generates a random number whose value range is $[0, 1)$ (Line 3). If the generated random number is less than the optimal allocation ratio (Line 4), the memory allocator selects the GPU memory node as the target node (Line 5).[11] Otherwise, the memory allocator selects the CPU memory node as the target node (Line 7).

[11]Note that the memory allocator always selects the GPU memory node as the target node if the target application is classified as a bandwidth non-intensive application because the optimal allocation ratio is 1 (Equation 4).

The memory allocator checks if the selected target node is currently full (Line 9). If it is full and the other memory is not full (Line 11), the other memory node is selected as the target node (Line 12). Finally, the memory allocator places a page in the target node (Line 15).[12]

5.2.4 Discussion

We discuss the maximum performance gain of BLPP over the state-of-the-art technique (i.e., BAP) presented in [17] and the local memory placement policy, which only employs GPU memory. First, we investigate the maximum performance gain of BLPP over the local policy with a bandwidth-sensitive application, whose total execution time is largely determined by the data (D) transfer time between the SIMT cores and the memory nodes. The memory allocation ratios determined by BLPP and the local policy are $p_{OPT,S}$ (Equation 2) and $p_{Local} = 1$. The total execution time of BLPP and the local policy is computed as follows – $t_{BLPP} = \frac{p_{OPT,S} \cdot D}{B_G}$ and $t_{Local} = \frac{p_{Local} \cdot D}{B_G}$.

Therefore, the maximum performance gain of BLPP over the local policy is computed using Equation 7 for bandwidth-sensitive applications. With bandwidth-sensitive applications, we find that the performance gain of BLPP over the local policy increases as the bandwidth of the CPU memory increases.

$$g_B = \frac{t_{Local}}{t_{BLPP}} = \frac{B_G + B_C}{B_G} \tag{7}$$

We investigate the maximum performance gain of BLPP over BAP [17] for latency-sensitive applications. We assume a latency-sensitive application whose working-set size is smaller than the capacity of the GPU physical memory. Because BLPP allocates all pages in the GPU memory node, the effective latency of each memory request with BLPP is computed as follows – $L_{BLPP} = L_G$. In contrast, since BAP allocates pages with a memory allocation ratio of p_{BAP} ($= \frac{B_G}{B_G+B_C}$), the effective latency of each memory request with BAP is computed as follows – $L_{BAP} = \frac{B_G}{B_G+B_C} \cdot L_G + \frac{B_C}{B_G+B_C} \cdot L_C$.

Therefore, the maximum performance gain of BLPP over BAP for latency-sensitive applications is computed using Equation 8. With latency-sensitive applications, we observe that the performance gain of BLPP over BAP increases as $\frac{L_C}{L_G}$ increases.

$$g_L = \frac{L_{BAP}}{L_{BLPP}} = \frac{\frac{B_G}{B_G+B_C} \cdot L_G + \frac{B_C}{B_G+B_C} \cdot L_C}{L_G} \tag{8}$$

5.3 Evaluation

This section presents the results of a quantitative evaluation of BLPP. Specifically, we aim to investigate the following – (1) the performance impact of BLPP, (2) the performance comparison of the static and dynamic versions of BLPP, and (3) the performance sensitivity of BLPP to the bandwidth ratio of the GPU to CPU memory.

[12] If the target node is still full (e.g., both memory nodes are currently full), a page existing in the target node is replaced with the requested page by the LRU page replacement policy (omitted for conciseness).

Table 6: Simulation parameters

Parameter	Value
SIMT core	Core count: 15, SIMT width: 32, pipeline depth: 5
Per-core resource	Number of registers: 32768, scratchpad: 48KB, MSHRs: 64, warps: 48, threads: 1536
Schedulers	Warp scheduler: Greedy-Then-Oldest, CTA scheduler: round-robin
L1 data cache	Capacity: 16KB/core, line size: 128B, associativity: 4
Interconnect	Channel width: 32
L2 cache	Capacity: 1024KB, banks: 16, line size: 128B, associativity: 8
GPU memory	Total bandwidth (default): 200GB/s
GPU-CPU link	Latency: 100 cycles [17, 18]
CPU memory	Total bandwidth (default): 80GB/s

5.3.1 Experimental Methodology

We use the GPGPU-Sim simulator (version 3.2.2) [16], which we have extended to model the heterogeneous memory hierarchies, selective caching protocol [19], and BLPP. Table 6 summarizes the architectural parameters of the simulated GPGPU architecture, which are based on the configuration file in the GTX480 directory. Unless stated otherwise, the bandwidths of the GPU and CPU memory are set to 200GB/s and 80GB/s respectively, and the GPU-CPU link latency is set to 100 cycles. These settings are identical to those used in prior works [17, 18].

Table 7 summarizes all the evaluated benchmarks (i.e., `2MM`, `GE` (GEMM), `LM` (lavaMD), `SC` (scan), `ST` (streamcluster), `2DC` (2DCONV), `BFS`, `CFD`, and `II`), which are selected from the benchmark suites presented in [28, 35–37]. The memory-related performance data of each benchmark is collected by executing it only using the GPU memory node.[13] The evaluated benchmarks are largely classified into two categories (i.e., bandwidth-sensitive and other benchmarks) based on the classification criteria discussed in Section 5.2.2.

To compare the performance of different memory placement techniques quantitatively, each benchmark is executed with the following five page placement techniques – (1) local (GPU memory only), (2) static best (SB), which uses the optimal memory allocation ratio determined through extensive offline profiling (i.e., 20 different memory allocation ratios), (3) the state-of-the-art memory placement technique (BAP) proposed in [17], (4) static BLPP (S-BLPP), and (5) dynamic BLPP (D-BLPP).

[13]The memory traffic (i.e., B_M in Table 7) of some benchmarks is higher than 200GB/s because it captures all data traffic transmitted through the interconnection network (ICN), which includes the memory requests that are served by the L2 cache slices in the GPU memory partitions.

Table 7: Evaluated benchmarks

Category	Benchmark	$m_{G,L1D}$	$m_{G,L2}$	B_M (GB/s)
BW sensitive	2MM [28]	0.53	0.24	227.5
	GE [28]	0.57	0.36	222.3
	LM [35]	0.57	0.00	200.9
	SC [36]	0.62	0.40	149.0
	ST [35]	0.60	0.76	162.7
Others	2DC [28]	0.23	0.54	188.0
	BFS [35]	0.35	0.36	112.5
	CFD [35]	0.20	0.71	78.3
	II [37]	0.03	0.19	33.0

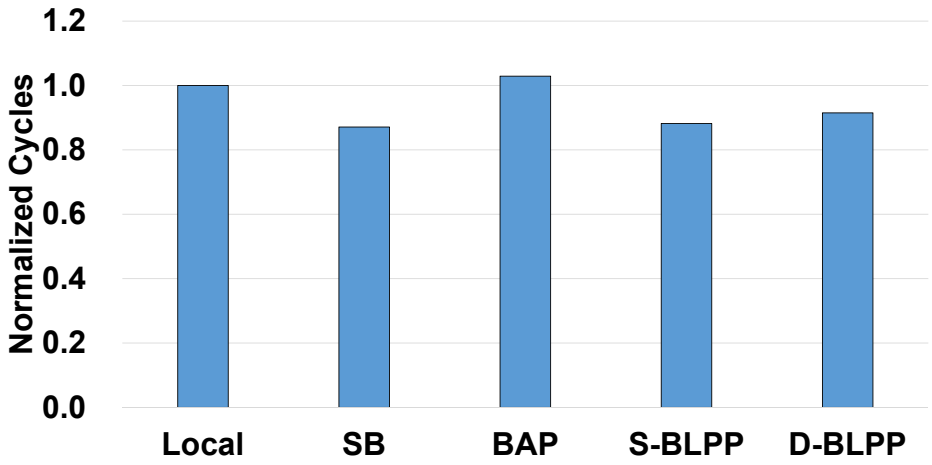


Figure 38: Overall performance results

5.3.2 Performance

First, we investigate the performance impact of BLPP. Figure 38 shows the average (i.e., geometric mean) execution cycles of each memory management technique normalized to that of the local version. We observe the following data trends. First, BLPP outperforms the local and BAP versions considerably. For example, S-BLPP achieves 13.4% and 16.7% higher performance compared to the local and BAP versions, respectively. This is mainly because BLPP accurately identifies the characteristics of the target application and effectively utilizes the GPU and CPU memory by preserving the optimal memory allocation ratio determined based on the application characteristics.

Second, BLPP achieves performance similar to that of the static best version, which requires extensive offline profiling. For instance, S-BLPP shows 1.2% lower performance than the static best version. This indicates that BLPP effectively utilizes the heterogeneous memory nodes by accurately identifying the characteristics of the target application.

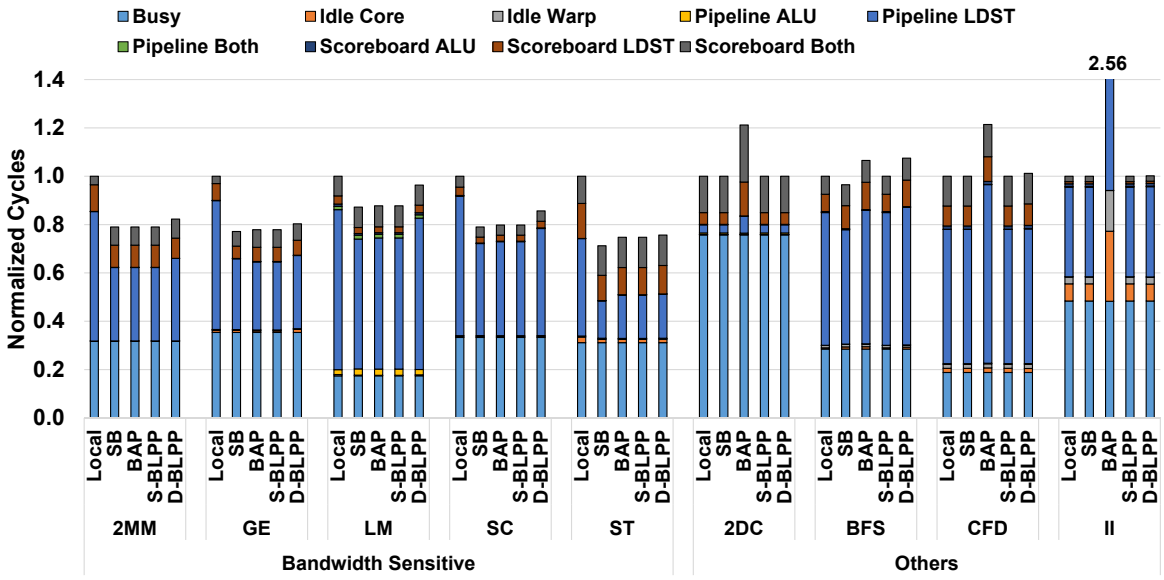


Figure 39: Execution cycle breakdowns

Third, S-BLPP and D-BLPP achieve comparable performances. Specifically, S-BLPP achieves 2.7% higher performance than D-BLPP. D-BLPP achieves performance comparable to S-BLPP because it robustly identifies the characteristics of the target application based on the performance counter data collected at runtime and determines the efficient memory allocation ratio. Our experimental results demonstrate that BLPP can be effectively employed as a runtime approach, thus eliminating the need for offline profiling.

5.3.3 In-depth Analysis

To gain deeper insight into the performance results, Figure 39 shows detailed cycle breakdowns of the evaluated benchmarks. Each bar is normalized to the local version and comprises multiple segments, each of which denotes the busy cycles (Busy), idle cycles due to a load imbalance across SIMT cores (Idle Core), idle cycles spent when no warp is ready to execute (Idle Warp), cycles spent for ALU (Pipeline ALU), LDST (Pipeline LDST), and both (Pipeline Both) pipeline stalls, and cycles stalled at the scoreboard to wait for the data produced by ALU (Scoreboard ALU), LDST (Scoreboard LDST), and both (Scoreboard Both) instructions.

In addition, Figure 40 shows the allocation ratio of the GPU memory for each of the evaluated benchmarks. Figures 39 and 40 demonstrate the following data trends. First, the performance gains of BLPP over the local and BAP versions are mainly achieved by reducing the cycles spent for LDST pipeline stalls (i.e., Pipeline LDST). This is because BLPP effectively utilizes all the available bandwidth of the heterogeneous memory by actively employing both the GPU and CPU memory nodes when executing bandwidth-sensitive benchmarks and reduces the memory latency by only employing the GPU memory node when executing the other benchmarks.

Second, the performance gains of BLPP over the local and BAP versions are also achieved

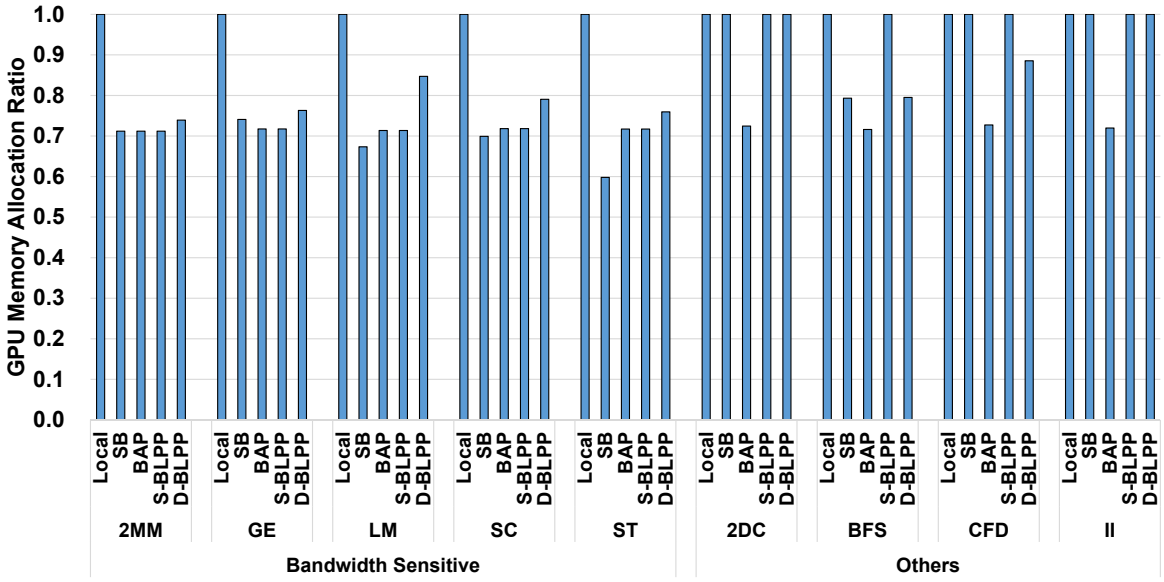


Figure 40: GPU memory allocation ratio

by reducing the cycles stalled at the scoreboard to wait for the data produced by LDST (i.e., Scoreboard LDST) instructions. This is also done because BLPP effectively utilizes the GPU and CPU memory nodes based on the characteristics of the target application, reducing the stalled cycles due to data hazards.

Third, interestingly, while the memory traffic of `2DC` is rather high, BAP is significantly outperformed by BLPP with `2DC`. This occurs because `2DC` effectively utilizes the L1 data cache (i.e., $m_{G,L1D} = 0.23$ in Table 7). While BAP employs additional bandwidth from the CPU memory node by allocating a subset of the memory pages in the CPU memory node, accesses to the pages allocated in the CPU memory node are disallowed to be cached in the L1 data cache [19], significantly degrading the overall performance of `2DC`, which effectively utilizes the L1 data cache. In contrast, BLPP accurately identifies the effective cache utilization of `2DC` and accordingly determines the optimal allocation ratio (i.e., $p_{OPT} = 1$), significantly outperforming BAP.

Fourth, in addition to the aforementioned performance gains, BLPP achieves additional performance gains over BAP for `II` by reducing the idle cycles due to a load imbalance across SIMT cores (i.e., Idle Core). This mainly occurs because the thread blocks scheduled on a set of SIMT cores access the CPU memory more frequently than those on the other set of SIMT cores, eventually causing a imbalanced execution cycles among the SIMT cores. In contrast, BLPP only employs the GPU memory by identifying `II` as a bandwidth insensitive application, eliminating the load imbalance.

Fifth, all the versions achieve similar performance with `BFS`. As shown in Table 7, `BFS` exhibits not only the relatively effective L1 data cache utilization but also the relatively high memory traffic. Thus, all the versions achieve similar performance regardless of their different allocation

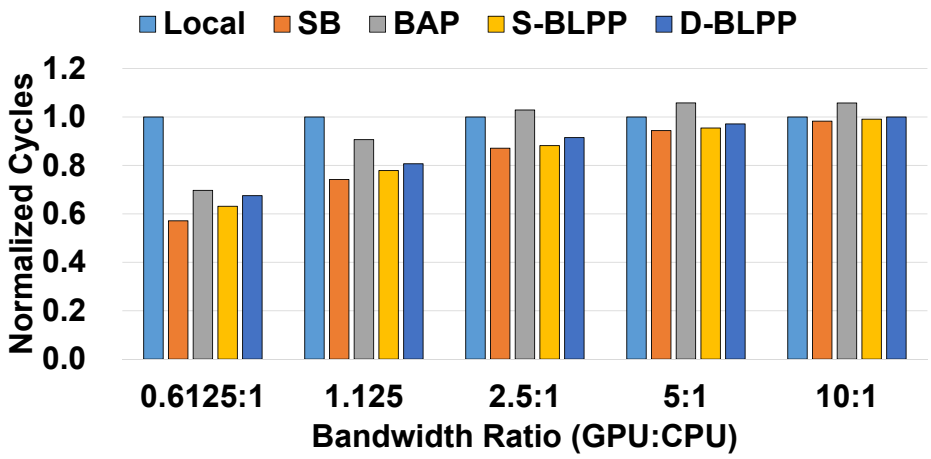


Figure 41: Performance sensitivity to the memory bandwidth ratio

ratios.

Finally, while D-BLPP shows performance comparable to that of S-BLPP on average across the benchmarks, D-BLPP achieves slightly lower performance than S-BLPP for some benchmarks (e.g., `LM` and `SC`). This is mainly because D-BLPP allocates pages only in the GPU memory before collecting the performance data of the target application, resulting in a higher GPU memory allocation ratio than the optimal allocation ratio (Figure 40). Nevertheless, D-BLPP achieves considerably higher performance than the BAP and local versions by dynamically finding the efficient allocation ratio and allocating pages across the heterogeneous memory nodes, showing that BLPP can be effectively implemented as a practical runtime system.

5.3.4 Sensitivity to Bandwidth Ratio

We investigate the performance sensitivity of BLPP to the bandwidth ratio of the GPU and CPU memory. Figure 41 shows the average (i.e., geometric mean) execution cycles of each memory management technique, normalized to those of the local version as the bandwidth ratio of the GPU to CPU memory is swept from 0.625 to 10. We observe the following data trends.

First, BLPP achieves high performance across all the bandwidth ratios by effectively utilizing the heterogeneous memory, demonstrating the robustness of BLPP. Second, as the bandwidth ratio decreases (i.e., the bandwidth gap between the GPU and CPU memory decreases), BLPP and BAP considerably outperform the local version by actively utilizing the CPU memory. Third, as the bandwidth ratio increases (i.e., the bandwidth gap between the GPU and CPU memory increases), the performance gap between BLPP and the local versions decreases as the performance gain that can be achieved by employing the CPU memory node decreases.

5.3.5 Summary of the Evaluation Results

Overall, our quantitative evaluation demonstrates the effectiveness of BLPP in that it considerably outperforms the state-of-the-art technique, achieves the performance similar to that of static best version, which requires extensive profiling, and delivers robust performance across various GPU-CPU memory bandwidth ratios.

VI Related Work

In the area of high performance and energy efficient CPU computing, advanced cache indexing (AC) has been actively researched. The proposed techniques can be largely classified into static [9–11] and adaptive [32] indexing schemes.

Static cache indexing: Static cache indexing schemes employ sophisticated hash functions to distribute memory addresses to different cache sets more evenly than the conventional indexing scheme [9–11]. Some of them build upon rather simple hash functions (e.g., bit-wise XOR [10]), suitable for L1 caches because of their simple and fast hardware implementation. Others use more complicated hash functions (e.g., prime modulo indexing [11]), targeting shared caches to minimize conflict misses at the cost of increased latency and hardware complexity. The main drawback of the static indexing schemes is that they lack dynamic adaptation based on the memory access pattern observed at runtime, potentially leading to suboptimal performance.

Adaptive cache indexing: To address the limitation of the static cache indexing, recent research has proposed an adaptive cache indexing scheme (ASCIB) [32]. ASCIB monitors the memory access pattern of workloads at runtime, determines the best indexing bits that are expected to minimize conflict misses for the observed memory access pattern, and periodically reconfigures them accordingly. In addition, since the hardware logic required for ASCIB is fully decoupled from the critical path for L1 cache accesses, ASCIB can be used for L1 caches. In this work, we adopt ASCIB as the baseline to implement the adaptive indexing scheme for GPGPU computing.

GPGPU cache indexing: Researchers have recently proposed ACI schemes for GPGPU architectures [6, 7]. These proposals build upon rather sophisticated static techniques – the arbitrary modulus indexing [7], polynomial modulus indexing [6], and full permutation-based indexing [46], which is a variant of the bit-wise XOR indexing. While insightful, as the case with CPU caches, the proposed techniques may lead to suboptimal performance and energy efficiency due to the lack of runtime adaptation. Our work is different from these proposals in the sense that we aim to investigate the effectiveness of the ACI schemes including both static and adaptive techniques for both the L1 data and L2 caches.

Adaptive GPGPU cache management techniques: Prior work has proposed adaptive GPGPU cache management techniques such as advanced cache indexing [6, 7, 12], warp scheduling [3–8, 47, 48], and cache bypassing [4, 5, 48–50]. While insightful, the GPGPU architectures proposed by most of the prior work integrates a subset of the adaptive cache management techniques. For instance, the architecture proposed in [4] lacks the advanced cache indexing technique, which is highly effective for mitigating GPGPU cache contention.

Khairy et al. have proposed an architectural framework (i.e., `DWT-PRIC`) that combines cache management techniques such as advanced static cache indexing and warp limiting [6]. Our work differs in that it investigates a fully integrated architectural framework for state-of-the-art adaptive cache management techniques including adaptive cache indexing. Further, in contrast

to DWT-PRIC, IACM employs adaptive cache management techniques in a more fine-grained and seamless manner (e.g., L1 data cache and bypassing can be simultaneously enabled). As demonstrated by our quantitative evaluation, IACM considerably outperforms DWT-PRIC (i.e., 361.4% at maximum and 7.7% on average) across all the evaluated benchmarks due to the aforementioned reasons.

Compiler-based static techniques for GPGPUs: Prior work has investigated compiler-based static techniques for GPGPUs such as cache bypassing [51], efficient memory layout [52], and parallelism management [53]. The proposed techniques identify the data access patterns of the target GPGPU application by analyzing its source code at compile time and generate the optimized binary for the target GPGPU application by injecting memory instructions that bypass the cache hierarchy [51] to mitigate cache contention, producing the efficient layout of the memory objects to effectively [52], or merging the code segments that are written to be executed by multiple threads into the single code segment that is executed by a single thread to manage the concurrency level [53]. While insightful, the proposed compiler-based static techniques have fundamental limitations in that their analysis fails to achieve high accuracy if the target application exhibits widely different behaviors based on the runtime conditions (e.g., input data size and type) or employs irregular data structures (e.g., graphs, trees), which leads to suboptimal efficiency.

In contrast, IACM dynamically performs adaptations based on the runtime information of the target GPGPU application, achieving high performance and energy efficiency as quantified in this work. Further, we believe that compiler-based static techniques can be incorporated into adaptive GPGPU cache management techniques such as IACM to further improve their efficiency. For instance, compiler-based static techniques can be used to determine the (potentially) efficient initial system states (e.g., the efficient initial warp count) and/or eliminate (potentially) suboptimal system states (e.g., low warp counts if the target application is expected to be scalable) from the system state space explored by adaptive cache management techniques.

CPU cache management techniques: Recent proposals have investigated the techniques that can effectively increase cache associativity with small performance overheads [39–41]. It is interesting future work to adopt these techniques in GPGPU caches. As shown in our quantitative evaluation, we believe that IACM can be used as an effective and complementary technique to further improve the performance and energy efficiency of highly associative GPGPU caches.

GPGPU heterogeneous memory management techniques: Prior works have presented the design and implementation of the virtual memory systems for GPGPUs with heterogeneous memory [17–21]. While insightful, most of the prior works have investigated the design and implementation of the efficient address translation [20, 21], cache hierarchy [19], and warp scheduling [18] with little or no focus on heterogeneity-aware memory management. Our work differs in that it investigates the efficient memory management technique for GPGPUs with heterogeneous memory, which robustly places pages across the heterogeneous memory nodes based on their performance differences.

The prior work closely related to ours is the work presented in [17]. The state-of-the-art technique (i.e., BAP) proposed in [17] places memory pages by considering the bandwidth difference of the GPU and CPU memory. While insightful, it lacks the consideration of the performance effects of the GPGPU caches and places memory pages in a latency-oblivious manner, achieving suboptimal performance as quantified in this work. In contrast, BLPP fully considers the performance effects of the GPGPU caches and allocates pages across the heterogeneous memory nodes in a bandwidth- and latency-aware manner, achieving considerably higher performance than BAP.

CPU heterogeneous memory management techniques: Prior works have presented the performance analysis and optimization of CPU-based systems with heterogeneous memory [45, 54–57] or NUMA systems [58–62]. While impactful, the prior works take into account only the latency [54] or bandwidth [45, 55, 56, 63] properties of heterogeneous memory nodes without simultaneously considering the both properties or mainly focus on the performance impact of the latency that incurs from remote memory nodes [57–62] for CPU-based systems. Our work differs in that it investigates the bandwidth- and latency-aware memory management technique in the context of GPGPU computing.

VII Conclusions

This dissertation proposes various methods to improve the performance and energy efficiency of GPGPU computing both in terms of computer architecture and system software by proposing adaptive cache and memory management techniques. We believe both hardware and software solutions must be provided in order to fully eliminate GPGPU performance pathologies. We provide quantitative evaluation for all of the techniques we propose and compare it with the baseline and state-of-the-art techniques.

First, this dissertation investigates the effectiveness of advanced cache indexing (ACI) for high-performance and energy-efficient GPGPU computing. We first explain performance pathologies of GPGPU hardware cache by analyzing memory access pattern of GPGPU application. Then, we discuss static indexing schemes (i.e., bit-wise XOR, polynomial modulus, and prime modulo indexing) and adaptive cache indexing schemes that can mitigate the issues. We discuss design and implementations of advanced cache indexing schemes to GPGPU hardware. Our quantitative evaluation demonstrates that the ACI schemes significantly improve performance and energy efficiency over the conventional indexing scheme across various GPGPU workloads. Our in-depth analysis shows that ACI reduces significant misses and reservation fails which leads to performance and energy efficiency improvement. In addition, our experimental results show that the ACI schemes continue to provide significant performance gains over the conventional indexing scheme even when the additional indexing latency occurs due to the hardware complexity and the baseline cache is enhanced with higher associativity and larger capacity. Overall, our quantitative evaluation demonstrates that the ACI schemes are promising for high performance and energy efficient GPGPU computing.

Second, this work presents IACM, integrated adaptive cache management for high-performance and energy-efficient GPGPU computing. IACM incorporates the state-of-the-art adaptive cache management techniques (i.e., adaptive cache indexing, adaptive warp limiting, and cache bypassing) in a unified architectural framework. We present three IACM designs, each with a different methods of unifying adaptive cache management techniques. We perform extensive design parameter sweeps to find parameters with the best performance for each of three IACM designs based on a cycle-accurate GPGPU simulator. We then compare performance of three IACM designs and determine the design that provides the highest performance among the three. Our quantitative evaluation demonstrates that IACM significantly outperforms the baseline architecture in terms of performance and energy efficiency (i.e., 98.1% and 61.9% on average, respectively), achieves considerably higher performance than the state-of-the-art technique (i.e., 361.4% at maximum and 7.7% on average), and provides significant performance and energy-efficiency gains over the baseline architecture enhanced with advanced architectural technologies such as higher associativity, larger capacity, and interconnect bandwidth. In summary, IACM is promising in that it significantly improves the performance and energy efficiency of various GPGPU benchmarks with the current technology and continues to deliver significant perfor-

mance and energy-efficiency gains even when the GPGPU memory hierarchy is enhanced with advanced technologies.

Finally, this dissertation proposes BLPP, bandwidth- and latency-aware page placement for GPGPUs with heterogeneous memory. BLPP is comprised of three phases. First, BLPP collects the performance characteristics (e.g., cache miss rates and memory traffic) of the target application using offline profile data or runtime information. Second, it determines the optimal allocation ratio across the heterogeneous memory nodes based on the application characteristics. Finally, it dynamically places memory pages by using the optimal allocation ratio determined from previous phase. Our experimental results show that BLPP is effective in that it considerably outperforms the state-of-the-art technique and achieves the performance similar to the static best version, which requires extensive offline profiling. We also propose two versions of BLPP, Static BLPP (S-BLPP), and Dynamic BLPP (D-BLPP). S-BLPP uses offline profile data of the target application while D-BLPP uses runtime data. Our quantitative evaluation demonstrates that D-BLPP achieves similar performance to S-BLPP, which shows that BLPP can be effectively implemented as a practical runtime system.

References

[1] "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.

[2] "AMD Graphics Cores Next (GCN) Architecture White Paper," Jun. 2012.

[3] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 72–83. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.16

[4] X. Chen, L.-W. Chang, C. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient gpu computing," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, ser. MICRO-47, Dec 2014, pp. 343–355.

[5] W. Jia, K. Shaw, and M. Martonosi, "Mrpb: Memory request prioritization for massively parallel processors," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, ser. HPCA '14, Feb 2014, pp. 272–283.

[6] M. Khairy, M. Zahran, and A. G. Wassal, "Efficient utilization of gpgpu cache hierarchy," in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU'15. New York, NY, USA: ACM, 2015, pp. 36–47. [Online]. Available: http://doi.acm.org/10.1145/2716282.2716291

[7] J. Diamond, D. Fussell, and S. Keckler, "Arbitrary modulus indexing," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, ser. MICRO-47, Dec 2014, pp. 140–152.

[8] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 308–317. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155656

[9] B. R. Rau, "Pseudo-randomly interleaved memory," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ser. ISCA '91. New York, NY, USA: ACM, 1991, pp. 74–83. [Online]. Available: http://doi.acm.org/10.1145/115952.115961

[10] A. González, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating cache conflict misses through xor-based placement functions," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. New York, NY, USA: ACM, 1997, pp. 76–83. [Online]. Available: http://doi.acm.org/10.1145/263580.263599

[11] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, ser. HPCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 288–. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2004.10015

[12] K. Y. Kim and W. Baek, "Quantifying the performance and energy efficiency of advanced cache indexing for gpgpu computing," *Microprocessors and Microsystems*, vol. 43, pp. 81 – 94, 2016, many-Core System-on-Chip Architectures and Applications (PDP 15). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933116000053

[13] K. Y. Kim, S. Kim, and W. Baek, "On the feasibility of advanced cache indexing for high-performance and energy-efficient gpgpu computing," in *Proceedings of the 3rd International Workshop on Many-core Embedded Systems*, ser. MES '15. New York, NY, USA: ACM, 2015, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2768177.2768179

[14] K. Y. Kim, J. Park, and W. Baek, "Iacm: Integrated adaptive cache management for high-performance and energy-efficient gpgpu computing," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, Oct 2016, pp. 380–383.

[15] K. Y. Kim, J. Park, and W. Baek, "Improving the performance and energy efficiency of gpgpu computing through integrated adaptive cache management," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 630–645, March 2019.

[16] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, ser. ISPASS '09, April 2009, pp. 163–174.

[17] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for gpus within heterogeneous memory systems," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 607–618. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694381

[18] M. Mao, W. Wen, X. Liu, J. Hu, D. Wang, Y. Chen, and H. Li, "Temp: Thread batch enabled memory partitioning for gpu," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 65:1–65:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2898103

[19] N. Agarwal, D. W. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler, "Selective GPU caches to eliminate CPU-GPU HW cache coherence," in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, ser. HPCA '16, 2016, pp. 494–506. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2016.7446089

[20] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 743–758. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541942

[21] S. Shahar, S. Bergman, and M. Silberstein, "Activepointers: A case for software address translation on gpus," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 596–608. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2016.58

[22] "Unified memory in cuda 6," https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/.

[23] "HSA Platform System Architecture Specification."

[24] K. Y. Kim and W. Baek, "Blpp: Improving the performance of gpgpus with heterogeneous memory through bandwidth- and latency-aware page placement," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct 2018, pp. 358–365.

[25] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, ser. HPCA '14, 2014, pp. 568–578. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2014.6835965

[26] "Nvidia nvlink high-speed interconnect," http://www.nvidia.com/object/nvlink.html.

[27] "Amd hypertransport^{TM}technology," http://www.amd.com/en-us/innovations/software-technologies/hypertransport.

[28] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *Innovative Parallel Computing (InPar), 2012*, ser. InPar '12, May 2012, pp. 1–10.

[29] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal, "A detailed gpu cache model based on reuse distance theory," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, ser. HPCA '14, Feb 2014, pp. 37–48.

[30] N. Topham, A. Gonzalez, and J. Gonzalez, "The design and performance of a conflict-avoiding cache," in *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, Dec 1997, pp. 71–80.

[31] Q. Yang and L. W. Yang, "A novel cache design for vector processing," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA '92. New York, NY, USA: ACM, 1992, pp. 362–371. [Online]. Available: http://doi.acm.org/10.1145/139669.140398

[32] A. Ros, P. Xekalakis, M. Cintra, M. E. Acacio, and J. M. García, "Ascib: Adaptive selection of cache indexing bits for removing conflict misses," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '12. New York, NY, USA: ACM, 2012, pp. 51–56. [Online]. Available: http://doi.acm.org/10.1145/2333660.2333674

[33] A. Ros, P. Xekalakis, M. Cintra, M. E. Acacio, and J. M. García, "Adaptive selection of cache indexing bits for removing conflict misses," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1534–1547, June 2015.

[34] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA'13. New York, NY, USA: ACM, 2013, pp. 487–498. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485964

[35] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306797

[36] "Cuda samples," http://docs.nvidia.com/cuda/cuda-samples/, 2016.

[37] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT'08. New York, NY, USA: ACM, 2008, pp. 260–269. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454152

[38] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735702

[39] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 187–198. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.20

[40] M. K. Qureshi, D. Thompson, and Y. N. Patt, "The v-way cache: Demand based associativity via global replacement," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 544–555. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2005.52

[41] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555778

[42] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 389–400. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.43

[43] "Cacti," http://www.hpl.hp.com/research/cacti/.

[44] "NVIDIA GF100: World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism."

[45] S. Yu, S. Park, and W. Baek, "Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: ACM, 2017, pp. 18:1–18:10. [Online]. Available: http://doi.acm.org/10.1145/3079079.3079092

[46] B. Wang, Z. Liu, X. Wang, and W. Yu, "Eliminating intra-warp conflict misses in gpu," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 689–694. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2755911

[47] A. ElTantawy and T. M. Aamodt, "Warp scheduling for fine-grained synchronization," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 375–388.

[48] Y. Liang, X. Xie, Y. Wang, G. Sun, and T. Wang, "Optimizing cache bypassing and warp scheduling for gpus," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 8, pp. 1560–1573, Aug 2018.

[49] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-driven dynamic gpu cache bypassing," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 67–77. [Online]. Available: https://doi.org/10.1145/2751205.2751237

[50] H. Dai, C. Li, H. Zhou, S. Gupta, C. Kartsaklis, and M. Mantor, "A model-driven approach to warp/thread-block level gpu cache bypassing," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016, pp. 1–6.

[51] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on gpus," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 516–523. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561929

[52] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, "User-defined distributions and layouts in chapel: Philosophy and framework," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 12–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863086.1863098

[53] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 86–97. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806606

[54] S. Kannan, A. Gavrilovska, and K. Schwan, "pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 13:1–13:16. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901325

[55] D. Shin and J. W. Lee, "Bandwidth-aware dram page migration for heterogeneous mobile memory systems," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*, Jan 2018, pp. 1–5.

[56] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 58:1–58:14. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126923

[57] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[58] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large pages may be harmful on numa systems," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 231–242. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643659

[59] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach, "Scalable task parallelism for numa: A uniform abstraction for coordinated scheduling and memory management," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: ACM, 2016, pp. 125–137. [Online]. Available: http://doi.acm.org/10.1145/2967938.2967946

[60] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen, "Numagic: A garbage collector for big data on big numa machines," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 661–673. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694361

[61] J. Park, M. Han, and W. Baek, "Quantifying the performance impact of large pages on in-memory big-data workloads," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '16, Sept 2016, pp. 1–10.

[62] J. Park and W. Baek, "Quantifying the performance and energy-efficiency impact of hardware transactional memory on scientific applications on large-scale numa systems," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 804–813.

[63] M. Han, J. Hyun, S. Park, and W. Baek, "Hotness- and lifetime-aware data placement and migration for high-performance deep learning on heterogeneous memory systems," *IEEE Transactions on Computers*, pp. 1–1, 2019.

Acknowledgements

First, I would like to thank my advising Professor Woongki Baek. He has taught me valuable skills during the Ph.D program, which helped me become not only a better researcher but also a better person in general. I will use these valuable skills throughout my professional career. He also believed in me and encrouged me to always do better. I could not have achieved any of my work without his help.

Second, I am thankful for the valuable feedback from the committee members, Professor Young-ri Choi, Professor Jongeun Lee, Professor Sam H. Noh, and Professor Kyuho Lee. Their feedbacks could help me improve the quality of this dissertation.

Third, I would also like to thank my fellow graduate students of both CASL and CISSR group. It was fortunate have fellow students to share research interests and student life. Thanks to them, Ph.D program has been a very fun process. I wish them luck to the rest of their graduate program.

Finally, I would like to give thanks to my family and friends. They have given me full support throughout the program.